VIGNETTE

# Cache On Delivery (COD)
# Cache Manager (CM)
# Design Document

Vignette Corporation

Two Barton Skyway

1601 South MoPac Expressway

Austin, TX 78746

Author: David Caldwell

Version 17

November 20, 2002

## CONFIDENTIAL

# Table of Contents

| Version | Date | Author | Remarks |
|---|---|---|---|
| 0.1 | 1/10/02 | David Caldwell | Initial version. |
| 0.2 | 3/25/02 | David Caldwell | Placement protocol and placement message handler sections are fairly solid. They are reviewed and feedback is incorporated. Other sections, beware. |
| 0.3 | 5/14/02 | David Caldwell | Added instance state diagram.<br>Added note to regenerate-last-accessed-interval about possible WS in memory caching problem.<br>Renamed physical cache section to cache manager and folded the physical cache's responsibilities into the cache manager (since they're 1-1).<br>Added closed issues; moved 1 open issue to closed.<br>Reworked placementMsgHandler pursuant to closed issues.<br>Added instance deleter & associated config vars.<br>Reworked placement-msg, placementMsgHandler, & associated diagrams with new include and action schemes.<br>Reject template-placements for app-server-only if cache is ws-only. |
| 0.4 | 5/15/02 | David Caldwell | Added concurrency architecture.<br>Removed placement rejection based on appServerOnly flag.<br>Added setting of expirationTimestamp.<br>Removed PAGE_GENERATOR_TYPE and WEB_SERVER_TYPE config vars. |
| 0.5 | 5/23/02 | David Caldwell | Added POOL_SIZE config var.<br>Renamed rcdDeployCount to recordStateGuid.<br>Changed placementMsgHandler's method of detecting concurrent placements from "instance insert duplicate row" to "file locked."<br>Added sections: AM-CM metadata-msg, AM-CM record-udpate-msg, MetadataMsgHandler, and PeriodicExpiration. |
| 0.6 | 5/31/02 | David Caldwell | Added open issue: Clearing WebserverLookup table when WS path to PG path mapping changes.<br>Added clear-cache-msg and handler sections.<br>Added dgId.<br>Maintain TemplateHash in TemplateState table.<br>Added ModuleInternalValue.<br>Removed timezone from expire-period. |
| 0.7 | 5/31/02 | David Caldwell | Added RecordUpdateMsgHandler |

| | | | |
|---|---|---|---|
| 0.8 | 6/10/02 | David Caldwell | Added Regenerator. Changed PlacementMsgHandler to:<br>• Handle regen placements<br>• Use higher-level ModuleValues metadata class<br>Refined MetadataMsgHandler.updateTemplateMetadata.<br>Added open issue: Exploded deployment and CDF external to .war file deployment order problem.<br>Added periodicExpirationTimestamp processing.<br>Added HOST, PORT, REGENERATE_HOST, & REGENERATE_PORT config vars.<br>Moved message receivers under generic framework.<br>Put some stuff in Monitoring section. |
| 0.9 | 6/12/02 | David Caldwell | Conform to new ModuleParams metadata-msg structure.<br>Handle placement-msg in 1st part of http msg rather than preamble + subsequent part lengths are now Content-length headers in each part. |
| 0.10 | 6/13/02 | David Caldwell | Minor formatting changes.<br>Increased CLEANUP_DELAY to 5 minutes. |
| 0.11 | 7/11/02 | David Caldwell | Corrected "PG Placement message to CM" diagram in section 4.3.<br>Added references section.<br>Named vgnId consistently in SSIs.<br>Corrected SSI offset in scenario in PlacementMsgHandler section. Added corresponding note in pseudo code.<br>Made CacheManager diagrams only show fields & methods used by calling handler.<br>Added CacheManager diagram to CM section with all fields and methods.<br>Fixed inconsistencies in a invalidateOrRegenEachInstance and policySaysRegenerate.<br>Component forward path changes: used component pathAtPg in SSI & regen request instead of fwd path; removed vgnComponent QS; added component regen headers.<br>PlacementMsgHandler handles wsLookup even on duplicate placements; also sets inst parentUrlAtWs.<br>Added dynamic-msg.<br>Added closed issues - WebServerLookup Concurrency, and Redundant metadata-msgs.<br>Corrected stage-cds-dg diagram and description in intro.<br>Added conditional transition to Regen state to Instance state diagram.<br>Deleted minheap icon from concurrency architecture diag.<br>Changed pathAtWs to urlAtWs. |

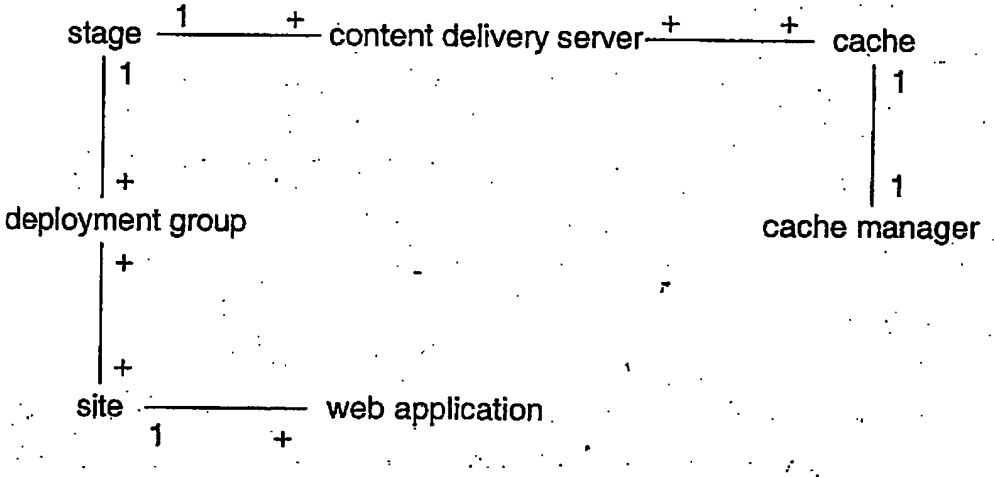
| | | | |
|---|---|---|---|
| | | | PlacementMsgHandler deletes cache file when concurrent expiration occurs.<br>Renamed DIR_ROOT to CACHE_DIR.<br>Noted that COMPONENT_SSI_PREFIX will be go away.<br>Made CLEANUP_PERIOD/DELAY config vars hidden.<br>Added type, min, & max for all numeric config vars.<br>CACHE_FILE_EXTENSION_SSI & CACHE_FILE_EXTENSION_NO_SSI - changed dynamic changes; moved to cache-level.<br>regenerate-last-accessed-interval - reference WS plugin doc for explain of WS last access oddities.<br>Added list of metadata clients to monitoring section.<br>Added open issues: "Too many cache files in a directory," "Configuration bootstrapping," "Configuration teardown," and "Failover."<br>CDS-level config vars are now at the stage:cds:site or stage:cds level.<br>Changed dgId to siteId. |
| 12 | 9/27/02 | David Caldwell | Switched version to match perforce revision.<br>Moved init routines from generic framework section to their object's sections.<br>Renamed dirRoot to cacheRoot.<br>Deleted dynamic-msg.<br>Renamed placement-msg.template-placement to cached-template.<br>Added optional placement-msg.dynamic-outer-template element before cached-template element.<br>Renamed Instance parentUrlAtWebserver to outerUrlAtWebserver.<br>If outerUrlAtWebserver is non-empty, use it as regen request uri for both outer templates and components. And for components, add x-vgn-stage-id, cds-id, site-id, component-path-at-pg, and response-content-type headers.<br>If outerUrlAtWebserver is empty (no WS tier), use path-at-pg for the regen request uri. And for components, do **not** add stage, cds, site, and path-at-pg headers.<br>Fixed processClearCacheMsg to handle path argument that maps to multiple templates. |
| 13 | 10/4/02 | David Caldwell | MetadataMsgHandler - changed to get msg from Http rather than JMS. Also added DaId for TemplateState table.<br>Plug into generic framework - added vgnagent/aglet info. |

| | | | |
|---|---|---|---|
| 14 | 11/7/02 | David Caldwell | Moved clearCacheOption argument to findTemplates call.<br>invalidateOrRegenInstance: made inst update a single trans.<br>Added CacheManager.getFullFilename.<br>Updated ClearCacheMsgHandler during implementation.<br>Updated RecordUpdateMsgHandler during implementation. |
| 16 | 11/11/02 | David Caldwell | Switched to version to **really** match perforce version.<br>Updated MetadataMsgHandler during delete & update implementation. |
| 17 | 11/20/02 | David Caldwell | Changed PlacementMsgHandler to make SSI & in MD component offsets be the offset in the original output rather than the offset in the cache file. |

# 1 Introduction

Cache manager (CM) caches and maintains dynamic file (template) output in a disk cache for use by a Web Server (WS) or a Page Generator (PG). Page generator is a generic term for the engine that executes a template (servlet engine, IIS instance, Tcl engine). The output is received from the PG filter in placement messages via Hypertext Transfer Protocol (HTTP).

The cached output of a template is called a cache file or a template instance.




A stage is associated with 1 or more deployment groups (DG). A DG is associated with only 1 stage.

A stage contains 1 or more Content Delivery Servers (CDSs). A CDS is in only 1 stage.

A site is a set of related content and web applications that progresses through stages of development. A stage is associated with 1 or more sites either through multiple DGs per stage or multiple sites per DG. Multiple DGs per stage are required when sites within a stage must be treated differently. An example is WS virtual hosting where each site's virtual domain has a separate docroot that receives only that site's content. A site is associated with 1 or more DGs, with each DG in a different stage.

A site has 1 or more web applications. A web application generates a single site.

A CM manages 1 cache. A cache can either be a WS cache, a PG cache, or both. A cache contains content from 1 or more stage, CDSs. A CDS's content can be cached in 1 or more caches.

A template is designated as cacheable by adding an entry to the Cache Definition File (CDF) that is deployed to the Application Manager (AM). There is 1 CDF per web app. AM sends per-template definition information to CM in metadata messages via the Persistent HTTP Messaging Framework [12].

CM saves the template-level cache definition information in the metadata database. As each template instance is cached, CM saves the corresponding instance-level metadata as well. Portions of the

template-level and instance-level metadata are replicated for read-only use by the WS plug-in and PG filter.

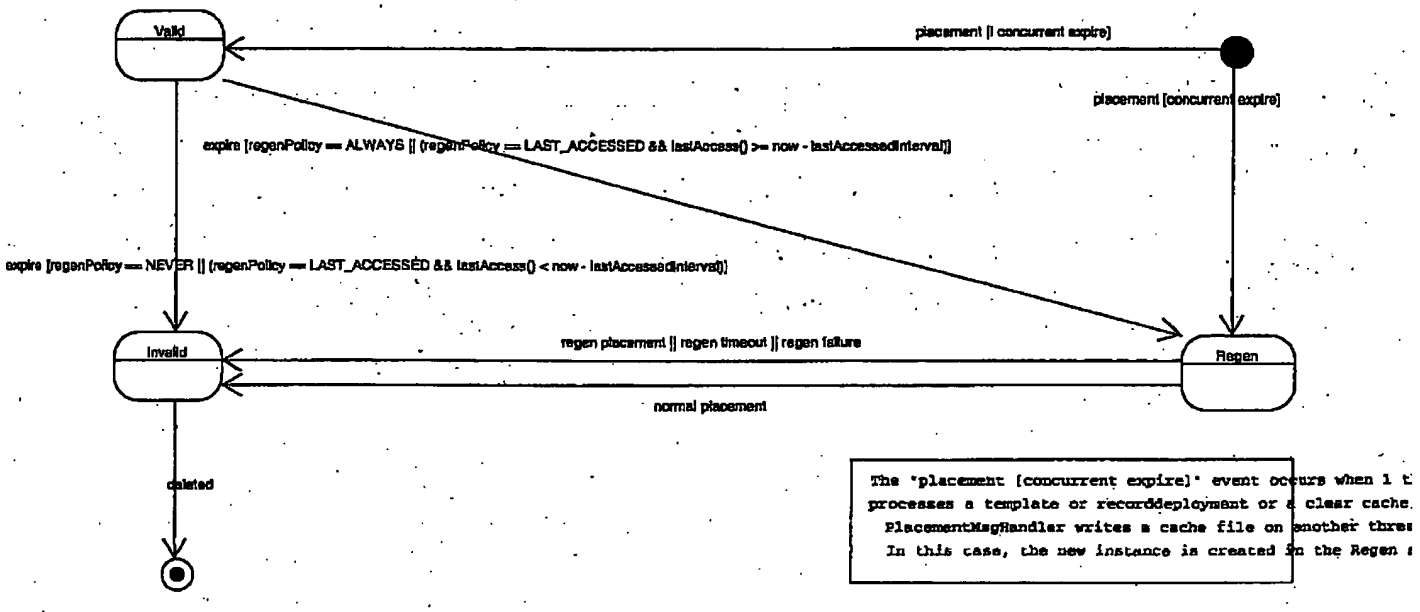
Expiration of a cache file means it is old. Depending on the regeneration policy, the file may be deleted, or the template may be re-executed to produce a new version of the cache file. Events that cause expiration are template update, periodic expiration, content change, or the clear cache API command.

Regeneration is the automatic re-execution of a template to produce a new version of a cache file. Regeneration is triggered by expiration. The regeneration request will contain the cache-sensitive request parameters saved from the request that originally caused the instance to be cached. Other parameters like the Content-type header are also included in the regeneration request.

Both expiration and regeneration are controlled by Policy Definition File (PDF) entries. There is 1 PDF file per site. AM sends per-template policy information to CM in metadata messages.

## *1.1 Instance state diagram*

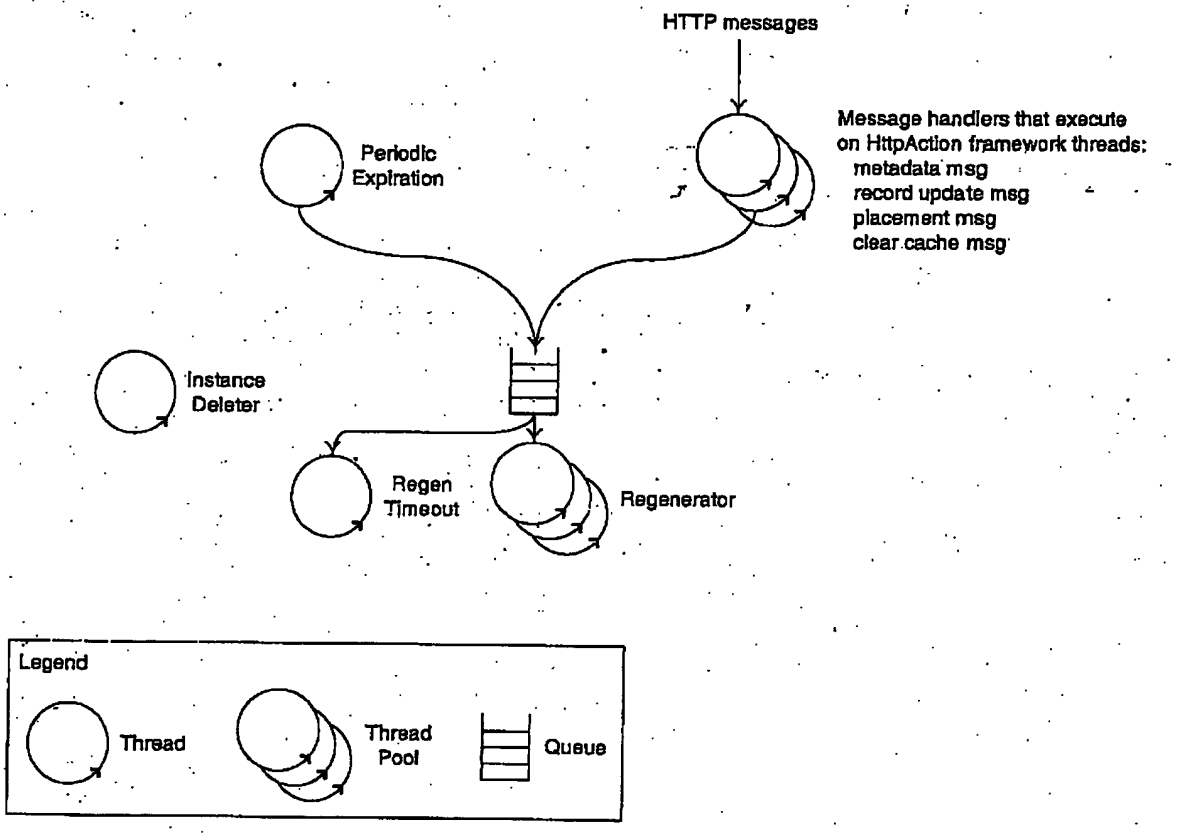
CM clients will only serve from the cache when the state is Valid or Regen.



# 2 Concurrency architecture

The following diagram shows the threads and thread pools that process the basic CM functions. Instance deleter and periodic expiration are single time-driven threads. Placement, template changes, record changes, and clear cache messages are all handled by the HttpAction framework thread pool whose size is determined by the POOL_SIZE configuration variable. Regeneration requests, responses, and placements are handled by a separate thread pool whose size is determined by the REGENERATE_CONCURRENCY_LIMIT configuration variable.

Periodic expiration, placement, template changes, record changes, and clear cache can all send regenerate messages to the regenerator pool.



# 3 Protocols

## *3.1 AM-CM metadata-msg*

The XML schema of the body of the HTTP metadata message is in ..\..\com\vignette\cod\cm\metadataMsg.xsd. The XML document describes the addition of templates, the deletion of templates, metadata changes, template changes, and simultaneous changes to a template and its metadata. MetadataMsgHandler on page 13 shows how the messages are handled.

## *3.2 AM-CM record-update-msg*

The XML schema of the body of the HTTP record update message is in ..\..\com\vignette\cod\cm\recordUpdateMsg.xsd. RecordUpdateMsgHandler on page 19 shows how the messages are handled.

## *3.3 PG-CM placement-msg*

The XML schema of the XML portion of the placement-msg is in ..\..\com\vignette\cod\cm\placementMsg.xsd. The XML document described by the schema is embedded in the 1st part of an HTTP 1.1 multipart/mixed message whose subsequent parts contain the template outputs to be cached. Each part has a Content-length header that specifies the # of bytes in the part. The XML document in the 1st part describes the template outputs in the other parts of the HTTP message and their interrelationships. PlacementMsgHandler on page 21 shows how the messages are handled.

## *3.4 PG-CM clear-cache-msg*

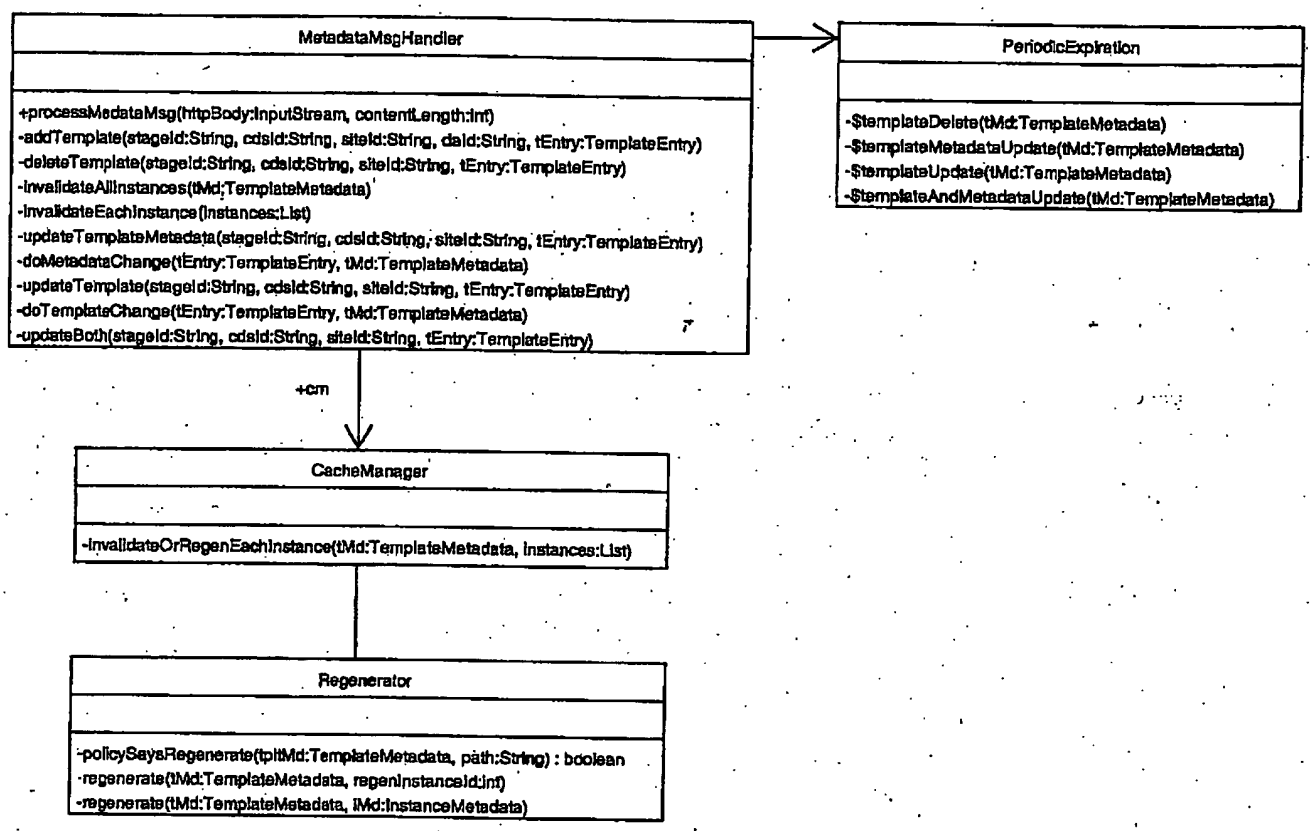
The XML schema of the body of the clear cache message is in ..\..\com\vignette\cod\cm\clearCacheMsg.xsd. The XML document described by the schema is embedded in either the body of an HTTP message. ClearCacheMsgHandler on page 32 shows how the messages are handled.

# 4 Message handlers

## *4.1 MetadataMsgHandler*

Handles the AM-CM metadata-msg described on page 12.

The following diagram shows the classes that process the metadata-msg:



The following pseudo-code shows how the metadata-msg is handled:

```
processMetadataMsg(httpBody : InputStream, contentLength : int)
   // Read the msg, then unmarshal it into a castor msg object
   byte[] buffer = new byte[contentLength]
   int bytesRead = httpBody.read(buffer)
   String string = new String(buffer, "UTF-8")
   StringReader stringReader = new StringReader(string)
   metadataMsg = MetadataMsg.unmarshalMetadataMsg(stringReader)

   String stageId = metadataMsg.getStageId
   String cdsId = metadataMsg.getCdsId
   String siteId = metadataMsg.getSiteId
   for each templateEntry in metadataMsg.getTemplateEntry,
      if templateEntry.getTemplateEvent == add,
         addTemplate(stageId, cdsId, siteId, metadataMsg.getDaId, templateEntry)
      else if templateEntry.getTemplateEvent == delete,
         deleteTemplate(stageId, cdsId, siteId, templateEntry)
      else if templateEntry.getTemplateEvent == update-metadata,
         updateTemplateMetadata(stageId, cdsId, siteId, templateEntry)
      else if templateEntry.getTemplateEvent == update-template,
         updateTemplate(stageId, cdsId, siteId, templateEntry)
      else if templateEntry.getTemplateEvent == update-both,
         updateBoth(stageId, cdsId, siteId, templateEntry)


addTemplate(stageId : String, cdsId : String, siteId : String, daId : String,
            tEntry : TemplateEntry)
   // Begin transaction
   mdCon.setAutocommit(false)

   // Ignore the metadata-msg if it's redundant
   if mdCon.templateExists(stageId, cdsId, siteId, tEntry.getPathAtPg), return

   insert Template metadata row tMd (and rows in associated tables like
         ModuleParameter and TemplateState) given the values in tEntry

   delete WebServerLookup rows with template's stageId, cdsId, siteId, and pathAtPg

   // End transaction
   mdCon.commit
   mdCon.setAutocommit(true)

   PeriodicExpiration.templateAdd(tMd)


deleteTemplate(stageId : String, cdsId : String, siteId : String,
               tEntry : TemplateEntry)
   // Begin transaction
   mdCon.setAutocommit(false)
```

```
    String pathAtPg = tEntry.getPathAtPg
    tMd = mdCon.getTemplateMetadata(stageId, cdsId, siteId, pathAtPg)

    // Ignore the metadata-msg if it's redundant
    if tMd == null, return
    if tMd.getObjectId != tEntry.getTemplateMetadata.getObjectId, return

    invalidateAllInstances(tMd)
    mdCon.deleteTemplate(tMd)

    delete WebServerLookup rows with template's stageId, cdsId, siteId, and pathAtPg

    // End transaction
    mdCon.commit
    mdCon.setAutocommit(true)

    PeriodicExpiration.templateDelete(tMd)


invalidateAllInstances(tMd : TemplateMetadata)
    instances = mdCon.findInstances(tMd)
    for each iMd in instances,
       iMd.setInstanceState(Invalid)
       iMd.setExpirationTimestamp(new Date)


invalidateEachInstance(instances : List)
    for each iMd in instances,
       mdCon.setTemplateInstanceState(iMd, Invalid)


updateTemplateMetadata(stageId : String, cdsId : String, siteId : String,
                       tEntry : TemplateEntry)
    tMd = mdCon.getTemplateMetadata(stageId, cdsId, siteId, tEntry.getPathAtPg)

    // Ignore the metadata-msg if it's redundant
    if tMd == null, return
    if tMd.getObjectId != tEntry.getTemplateMetadata.getObjectId, return
    if tMd.getModCount >= tEntry.getTemplateMetadata.getModCount, return
    if tMd.getMetadataHash == tEntry.getTemplateMetadata.getMetadataHash, return

    doMetadataChange(tEntry, tMd)


// Called when a metadata-msg indicates module metadata changed.
// Compares module parameters from template metadata to those in metadata-msg and
// invalidates instances according to the following rules:
//
//    if any parameters were added,
//       -> invalidate (no regen) all instances
//    if any sensitive or dependent parameters were removed,
//       -> invalidate (no regen) instances with ModuleValues for those parameters
//    if any no-cache parameters were removed,
```

```
//       -> just update metadata
//
// Note: here's an optimization that could be made to the case where sensitive or
// dependent parameters are removed.  Instead of deleting an instance X that uses a
// removed parameter P, Y_paramHash = X_paramHash - P.  If Y_paramHash is not in
// the instance metadata, add it and rename X's cache file from X_paramHash_v.ext
// to Y_paramHash_v.ext.
doMetadataChange(tEntry : TemplateEntry, tMd : TemplateMetadata)

   // Begin transaction
   mdCon.setAutocommit(false)

   // Invalidate all instances if any parameters were added.
   paramAdded = false
   for each nParam in tEntry.getTemplateMetadata.getModuleParams,
      found = false
      for each oParam in tMd.getModuleParameters,
         if nParam.getName == oParam.getName
               && nParam.getOptions == oParam.getOptions,
            found = true
            break
      if ! found,
         paramAdded = true
         break

   if ! paramAdded,

      // Invalidate instances that use any sensitive or dependent parameters that
      // were removed.
      for each oParam in tMd.getModuleParameters,
         found = false
         for each nParam in tEntry.getTemplateMetadata.getModuleParams,
            if nParam.getName == oParam.getName
                  && nParam.getOptions == oParam.getOptions,
               found = true
               break
         if ! found && oParam.getType != no-cache,
            removedParamList.add(oParam)
      if ! removedParamList.isEmpty,
         instances = mdCon.getInstancesUsingParams(tMd, removedParamList)

   update Template metadata row (and rows in associated tables like ModuleParameter
         and TemplateState) given the values in tEntry

   // End transaction
   mdCon.commit
   mdCon.setAutocommit(true)

   if invalidateAll,
      invalidateAllInstances(tMd)
   else if ! instances.isEmpty,
      invalidateEachInstance(instances)
```

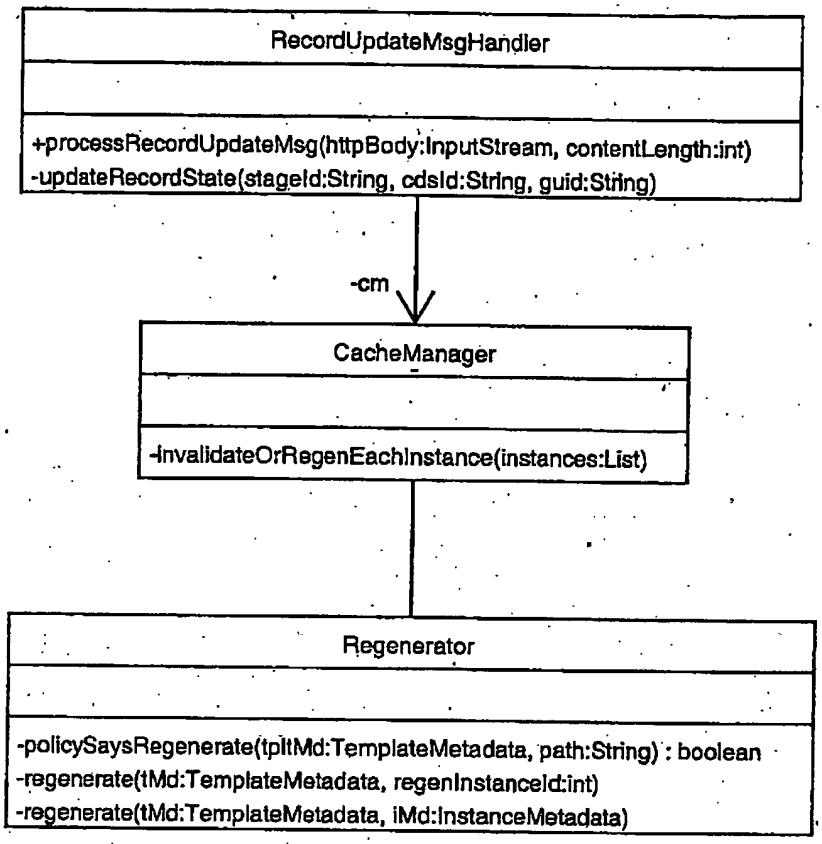
```
    PeriodicExpiration.templateMetadataUpdate(tMd)


updateTemplate(stageId : String, cdsId : String, siteId : String,
               tEntry : TemplateEntry)
    tMd = mdCon.getTemplateMetadata(stageId, cdsId, siteId, pathAtPg)

    // Ignore the metadata-msg if it's redundant
    if tMd == null, return
    if tMd.getObjectId != tEntry.getTemplateMetadata.getObjectId, return
    if tMd.getModCount >= mdMsg.getModCount, return
    if tMd.getTemplateHash == tHash, return

    doTemplateChange(tEntry, tMd)


doTemplateChange(tEntry : TemplateEntry, tMd : TemplateMetadata)
    String pathAtPg = tEntry.getPathAtPg

    // Begin transaction
    mdCon.setAutocommit(false)

    tHash = tEntry.getTemplateMetadata.getTemplateHash

    tMd.setTemplateHash(tHash)
    tState = tMd.getTemplateState(tEntry.getDaId)
    if tState == null,
       tState = new TemplateState
       tState.setDeploymentAgentId(tEntry.getDaId)
       templateStates = tMd.getTemplateStates
       templateStates.add(tState)
    tState.setTemplateHash(tHash)
    tMd.setObjectId(tEntry.getTemplateMetadata.getObjectId)
    tMd.setModCount(tEntry.getTemplateMetadata.getModCount)
    tMd.update

    // Get this template's valid instances.  Skip regen instances, since they've
    // already been been cleared by some other event.  If PG's already doing the
    // regen, the placement templateHash will control whether it's placed or regen'd
    // again.
    instances = mdCon.getAllInstances(tMd)

    // End transaction
    mdCon.commit
    mdCon.setAutocommit(true)

    cm.invalidateOrRegenEachInstance(tMd, instances)


updateBoth(stageId : String, cdsId : String, siteId : String,
           tEntry : TemplateEntry)
    tMd = mdCon.getTemplateMetadata(stageId, cdsId, siteId, tEntry.getPathAtPg)
```

```
// Ignore the metadata-msg if it's redundant
if tMd == null, return
if tMd.getObjectId != tEntry.getTemplateMetadata.getObjectId, return
if tMd.getModCount >= mdMsg.getModCount, return

if tMd.getMetadataHash != tEntry.getTemplateMetadata.getMetadataHash, return
   doMetadataChange(tEntry, tMd)

if tMd.getTemplateHash != tEntry.getTemplateMetadata.getTemplateHash, return
   doTemplateChange(tEntry, tMd)
```

## 4.2 *RecordUpdateMsgHandler*

Handles the AM-CM record-update-msg described on page 12.

The following diagram shows the classes that process the record-update-msg:

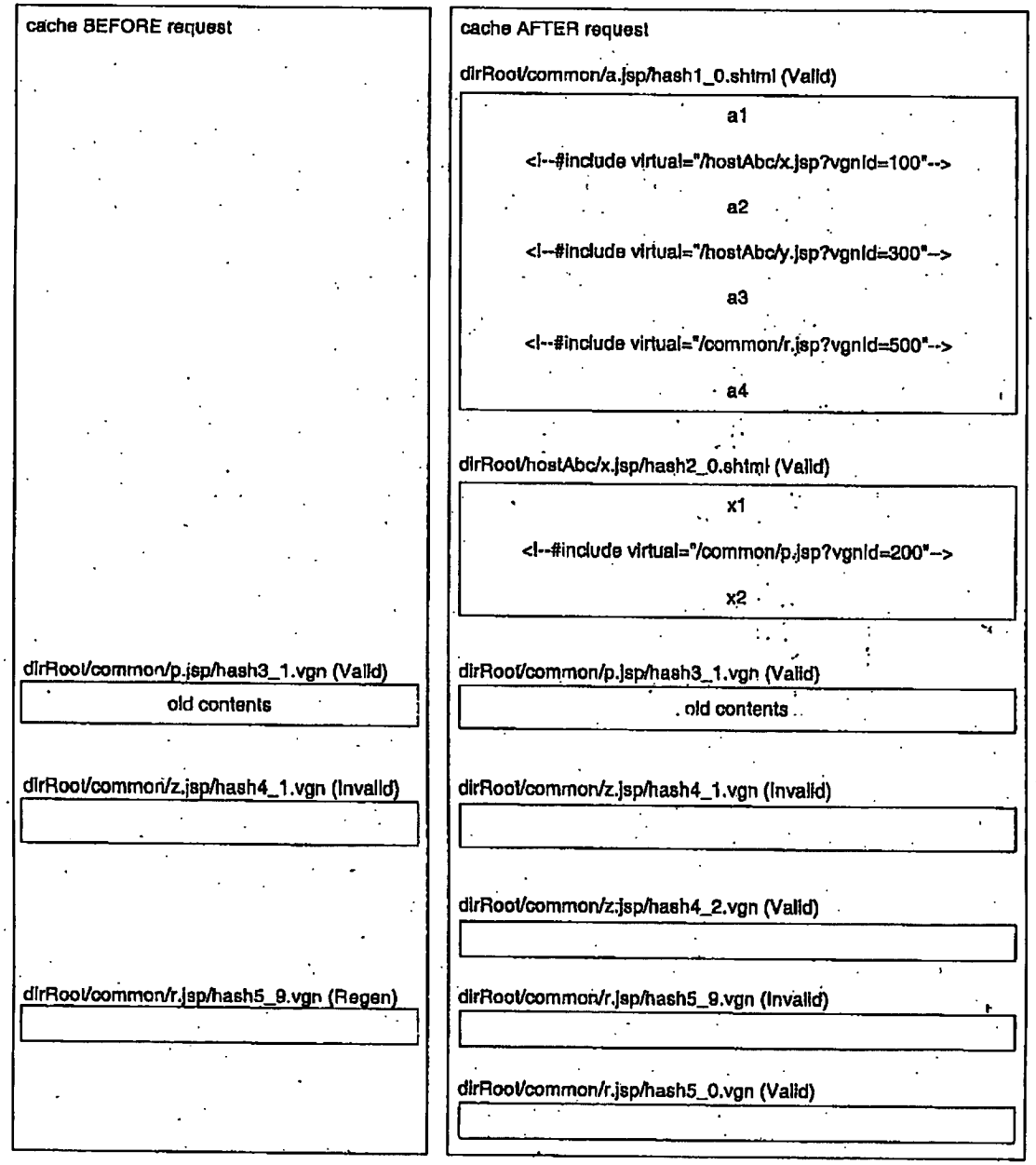


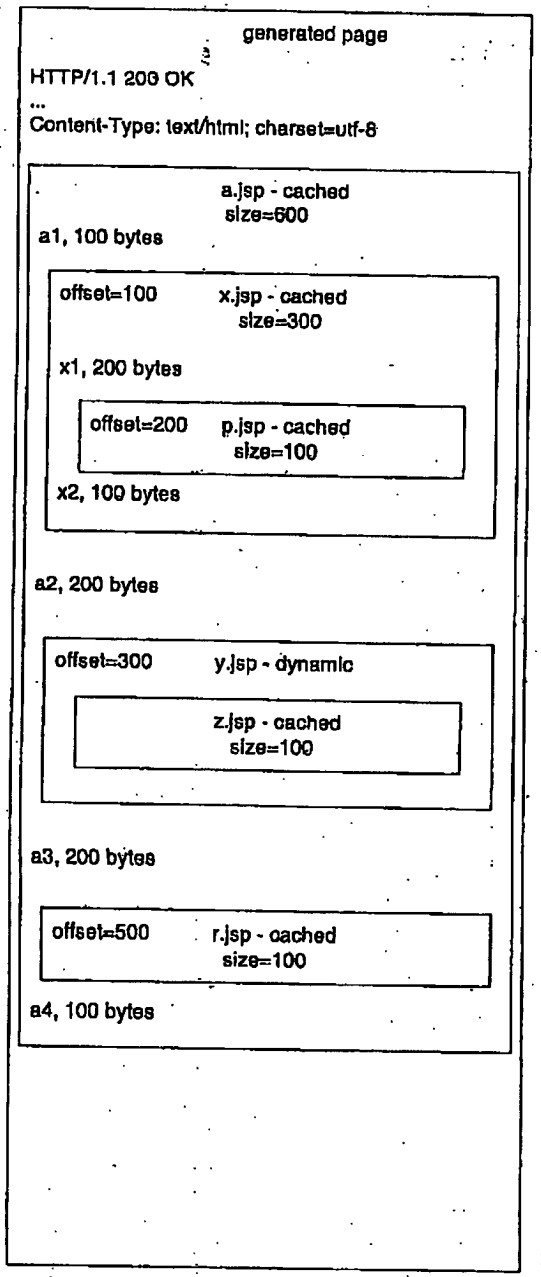
The following pseudo-code shows how the record-update-msg is handled:

```
processRecordUpdateMsg(httpBody : InputStream, contentLength : int)

   // Read the msg, then unmarshal it into a castor msg object
   byte[] buffer = new byte[contentLength]
   int bytesRead = httpBody.read(buffer)
   String string = new String(buffer, "UTF-8")
   StringReader stringReader = new StringReader(string)
   ruMsg = RecordUpdateMsg.unmarshalRecordUpdateMsg(stringReader)

   String stageId = ruMsg.getStageId
   String cdsId = ruMsg.getCdsId

   // Begin transaction
   mdCon.setAutocommit(false)

   // Get the valid instances affected by the updated and deleted records.
   // Skip regen instances, since if they're still waiting to go to PG, they'll get
   // the new data.  If PG's already doing the regen, the placement
   // record-state-guid will control whether it's placed or regen'd again.
   instances = mdCon.getInstancesByContentId(stageId, cdsId,
         ruMsg.enumerateRecordId)

   updateRecordState(stageId, cdsId, ruMsg.getGuid)

   // End transaction
   mdCon.commit
   mdCon.setAutocommit(true)

   cm.invalidateOrRegenEachInstance(instances)


updateRecordState(stageId : String, cdsId : String, guid : String)
   RecordState rcdState = mdCon.findRecordState(stageId, cdsId)
   rcdState.setGuid(guid)
   mdCon.update(rcdState)

   catch(RecordDoesNotExistException e)
      rcdState = new RecordState
      rcdState.setStageId(stageId)
      rcdState.setCdsId(cdsId)
      rcdState.setGuid(guid)
      mdCon.insert(rcdState)
```

### 4.3 *PlacementMsgHandler*

The scenario in the 2 following diagrams shows the major inputs and outputs of the placement message handler:

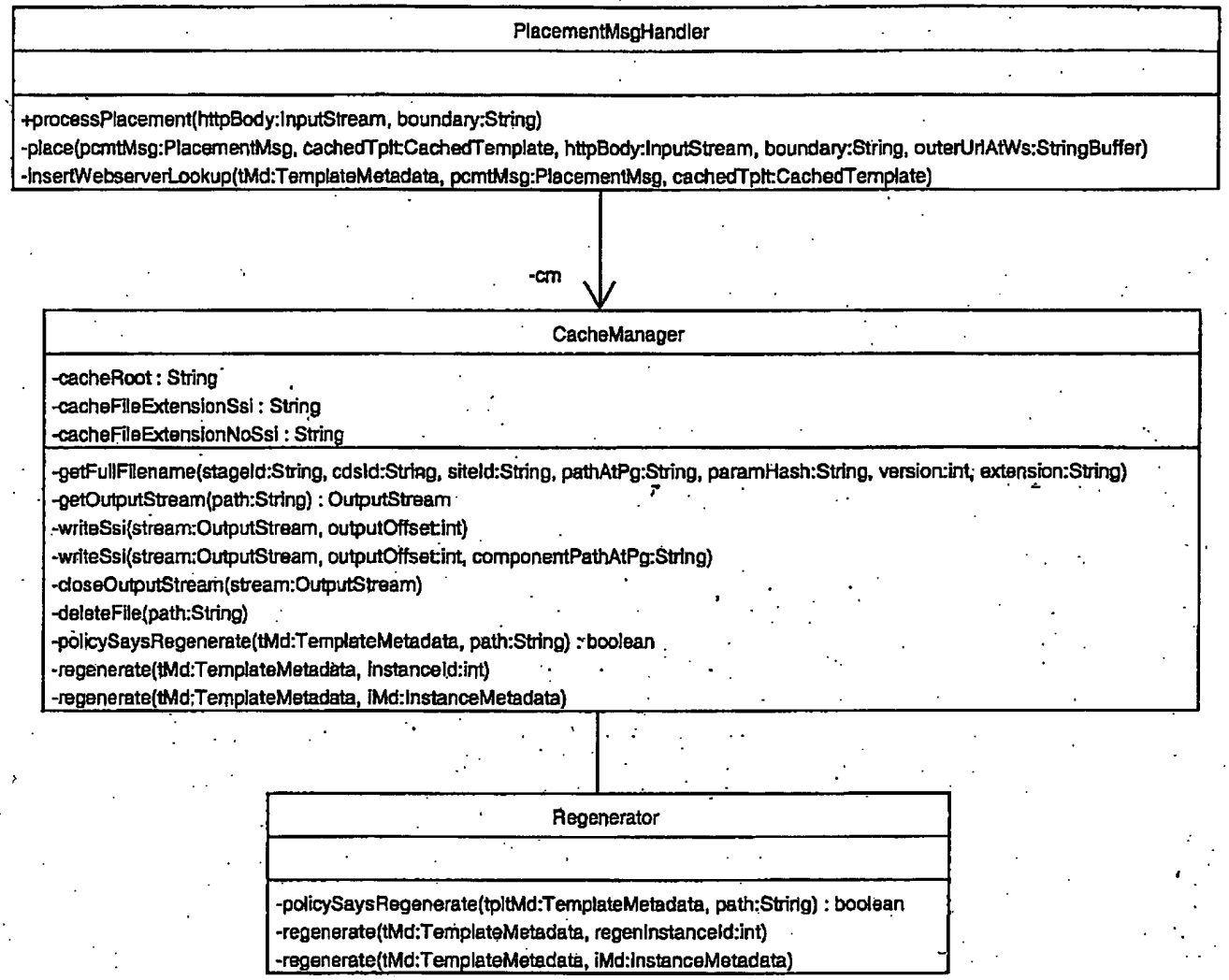





PG placement message to CM

```
POST http://cmHost:port/placement HTTP/1.1
...
Content-Type: multipart/mixed; boundary="unique-boundary-1"


--unique-boundary-1
Content-length: 841

<?xml version="1.0" encoding="UTF-8"?>
<placement-msg response-content-type="text/html; charset=utf-8" etc...>
   <template-placement path-at-pg="/common/a.jsp" param-hash="hash1" etc...>
      <include service="cod" type="component" offset="100">
         <param name="type" value="component"/>
         <param name="path-at-pg" value="/hostAbc/x.jsp"/>
      </include>
      <include service="cod" type="component" offset="300">
         <param name="type" value="component"/>
         <param name="path-at-pg" value="/hostAbc/y.jsp"/>
      </include>
      <include handler="cod" offset="500">
         <param name="type" value="component"/>
         <param name="path-at-pg" value="/common/r.jsp"/>
      </include>
   </template-placement>
   <template-placement path-at-pg="/hostAbc/x.jsp" param-hash="hash2">
      <include service="cod" type="component" offset="200">
         <param name="type" value="component"/>
         <param name="path-at-pg" value="/common/p.jsp"/>
      </include>
   </template-placement>
   <template-placement path-at-pg="/common/p.jsp" param-hash="hash3"/>
   <template-placement path-at-pg="/common/z.jsp" param-hash="hash4"/>
   <template-placement path-at-pg="/common/r.jsp" param-hash="hash5"/>
</placement-msg>
--unique-boundary-1
Content-length: 600

a.jsp output: a1, a2, a3, a4
--unique-boundary-1
Content-length: 300

x.jsp output: x1, x2
--unique-boundary-1
Content-length: 100

p.jsp output
--unique-boundary-1
Content-length: 100

z.jsp output
--unique-boundary-1
Content-length: 100

r.jsp output
--unique-boundary-1--
```

The following diagram shows the classes that process the placement-msg:




The following pseudo-code shows how the placement-msg is handled:

```
// httpBody points at beginning of body
processPlacement(httpBody : InputStream, boundary : String)
   size = getContentLengthOfNextPart(httpBody, boundary)
   read size bytes into buffer
   stringReader = new StringReader(buffer, "UTF-8")
   msg = PlacementMsg.unmarshalPlacementMsg(stringReader)
   outerUrlAtWs = null
   if msg.hasDynamicOuterTemplate,

      outer = msg.getDynamicOuterTemplate
      outerUrlAtWs = outer.getUrlAtWs

      // Begin transaction
```

```
        outer = msg.getDynamicOuterTemplate
        outerUrlAtWs = outer.getUrlAtWs

        // Begin transaction
        mdCon.setAutocommit(false)

        insert = true

        // If another placement-msg concurrently updated the table, don't insert
        if mdCon.hasWebServerLookup(outer.getUrlAtWs),
           insert = false

        // If a concurrent template add changed the template to cacheable,
        // don't insert
        if mdCon.hasTemplateMetadata(msg.getStageId, msg.getCdsId, msg.getSiteId,
              outer.getPathAtPg)
           insert = false

        if insert,
           WebServerLookup wsLookup = new WebServerLookup
           wsLookup.setUrlAtWs(outer.getUrlAtWs)
           wsLookup.setTemplateKey(null)
           wsLookup.setStageId(msg.getStageId)
           wsLookup.setCdsId(msg.getCdsId)
           wsLookup.setSiteId(msg.getSiteId)
           wsLookup.setPathAtPg(outer.getPathAtPg)
           mdCon.insertWebServerLookup(wsLookup)

        // End transaction
        mdCon.commit
        mdCon.setAutocommit(true)

     for each cachedTemplate in placementMsg,
        place(msg, cachedTemplate, httpBody, boundary, outerUrlAtWs)


  // Upon exit, next byte in httpBody is the data in the next part
  getContentLengthOfNextPart(httpBody : InputStream, boundary) : int
     read CRLF
     read -- boundary CRLF
     read "Content-length:"
     read length CRLF
     parse the length
     eat the CRLF that preceeds the body-part
     return the length


  // httpBody points at or just prior to next boundary
  place(placementMsg : PlacementMsg, cachedTplt : CachedTemplate,
        httpBody : InputStream, boundary : String, outerUrlAtWs : String)

     // If this is the outer cached page, remember the outer URL at WS to save in
     // included component's inst MD.  Do this 1st to get the url@Ws even if the
```

```
// outer template is ignored below.
if cachedTplt.getUrlAtWs != null && cachedTplt.getUrlAtWs.length != 0,
   outerUrlAtWs.replace(0, Integer.MAX_VALUE, cachedTplt.getUrlAtWs)

// Ignore the placement if CM processed a delete template while or after PG
// generated the page.
Metadata tpltMd = mdCon.getTemplateMetadata(placementMsg.getStageId,
      placementMsg.getCdsId, placementMsg.getSiteId, cachedTplt.getPathAtPg)
if tpltMd == null,
   return

// Ignore the placement if CM processed a metadata deployment after PG generated
// the page.  This can occur when a PG generates a placement while the template
// metadata is being updated.
if tpltMd.metadataHash != cachedTplt.metadataHash,
   return

// Ignore the placement if CM processed a template deployment after PG generated
// the page.  This can occur when a PG generates a placement while the template
// is being updated.
if tpltMd.templateHash != cachedTplt.templateHash,
   throw OldPlacementException

// Ignore the placement if CM processed a clear cache after PG generated the
// page.  This can occur when 1 PG processes clear cache while another generates
// a placement for an instance being cleared.
if tpltMd.clearCacheId != cachedTplt.clearCacheId,
   throw OldPlacementException

// Ignore the placement if CM processed a record deployment after PG generated
// the page.  This can occur when a PG generates a placement while a record
// is being updated.
RecordState rcdState = mdCon.getRecordState(placementMsg.getStageId,
      placementMsg.getCdsId)
if rcdState.getGuid != cachedTplt.getRecordStateGuid,
   throw OldPlacementException

// If a regeneratable expiration occurred during a regenerate,
// restart the regeneration.
catch OldPlacementException
   if cachedTplt.hasRegenInstanceId,
      cm.regenerate(tMd, cachedTplt.getRegenInstanceId)
   return

// Ignore the placement if it's a duplicate.  This occurs when 2 PGs process
// concurrent requests for the same instance.  It also occurs at preexisting
// caches when a new WS or PG cache is added to grow system capabilities.
Instance maxInst = mdCon.getMaxVersionOfInstance(tpltMd.getKey,
      cachedTplt.getParamHash)
if maxInst != null && maxInst.state == Valid,

   // PlacementMsgHandler needs to handle wsLookup info even if instance is
   // already cached (since there can be multiple url@ws per template, and since
```

```
    // WS can be added after instances are cached without parentUrl@Ws values)

    // If this is the outer cached page,
    if cachedTplt.hasUrlAtWs,

       // Start a metadata db transaction
       mdCon.setAutoCommit(false)

       insertWebServerLookup(tpltMd, placementMsg, cachedTplt)
       maxInst.setOuterUrlAtWs(cachedTplt.getUrlAtWs)
       mdCon.updateInstance(maxInst)

       // End transaction
       mdCon.commit
       mdCon.setAutocommit(true)

    else // component or direct PG request

       if outerUrlAtWs != null,
          maxInst.setOuterUrlAtWs(outerUrlAtWs)
          mdCon.updateInstance(maxInst)

    return

 // If there's instance metadata, increment its version
 version = 0
 if maxInst != null,
    version = maxInst.getVersion + 1

 // Determine the cache file extension
 if cachedTplt.getIncludeCount > 0,
    ext = cm.cacheFileExtensionSsi
 else
    ext = cm.cacheFileExtensionNoSsi

 // Build the cache filename: cacheRoot/stage/cds/site/pathAtPg/paramHash_v.ext
 path = cm.getFullFilename(stageId, cdsId, siteId, cacheTplt.getPathAtPg,
       cachedTplt.getParamHash, version, ext)

 // See if getOutputStream can use java.nio.channels.FileChannel lock methods to
 // only allow 1 thread to open the file at a time.
 stream = cm.getOutputStream(placementMsg.stageId, placementMsg.cdsId,
       placementMsg.getSiteId, path)

 // Abandon the placement if another thread created and locked the file just
 // before I did (see section 14.7.2.1 Problem 1 - concurrent open and write to
       the same file by different threads in the same process)
 catch (FileLockedException e)
    return

 size = getContentLengthOfNextPart(httpBody, boundary)

 inputIdx = 0
```

```
InstanceMetadata iMd = new InstanceMetadata
List includeMdList = iMd.getIncludes
for each include in cachedTplt,

   // Write the template output preceeding the include
   while (inputIdx < include.offset)
      byte[] buffer = new byte[include.offset - inputIdx]
      bytesRead = httpBody.read(buffer, 0, buffer.length)
      if bytesRead == -1
         close stream; delete cache file; log error
         throw PrematureEOF
      inputIdx += bytesRead

      stream.write(buffer, 0, bytesRead)

   // Remember the data (including the output offset) for each SSI written.
   // Store the ssi data in the instance metadata so the filter can skip
   // directly to the appropriate spot in the cache file during delivery rather
   // than having to parse the entire file.
   isComponentInclude = false
   componentPath = null
   Include includeMd = new Include
   includeMd.setHandler(include.getHandler)
   includeMd.setOffset(include.offset)
   List ipList = includeMd.getIncludeParameters
   for each param in include,
      IncludeParameter ip = new IncludeParameter
      ip.setName(param.getName)
      ip.setValue(param.getValue)
      ipList.add(ip)

      // cod include handler's component information is represented in the
      // placement-msg as follows:
      //
      // <include handler="cod" offset="100">
      //     <param name="type" value="component"/>
      //     <param name="path-at-pg" value="/hostAbc/x.jsp"/>
      // </include>
      if include.getHandler == "cod",
         if param.getName == "type" && param.getValue == "component",
            isComponentInclude = true
         else if param.getName == "path-at-pg",
            componentPath = param.getValue

   componentSsi = isComponentInclude && componentPath != null

   // Write the SSI
   ssiLength = componentSsi ? cm.writeSsi(stream, include.offset, componentPath)
                            cm.writeSsi(stream, include.offset)

   includeMd.setLength(ssiLength)
   includeMdList.add(includeMd)
```

```
// Write the template output that follows the last component
while (inputIdx < size)
   byte[] buffer = new byte[size - inputIdx]
   bytesRead = httpBody.read(buffer, 0, buffer.length)
   if bytesRead == -1
      close stream; delete cache file; log error
      throw PrematureEOF
   inputIdx += bytesRead

   stream.write(buffer, 0, bytesRead)

cm.closeOutputStream(stream)

// Start a metadata db transaction
mdCon.setAutoCommit(false)

iMd.setState(InstanceState.VALID)

// If another thread processed a metadata deployment while this thread wrote the
// cache file,
tpltMd = mdCon.getTemplateMetadata(placementMsg.getStageId,
     placementMsg.getCdsId, placementMsg.getSiteId, cachedTplt.getPathAtPg)
rcdState = mdCon.getRecordState(placementMsg.getStageId, placementMsg.getCdsId)
if tpltMd == null || tpltMd.metadataHash != cachedTplt.metadataHash,

   // Then the cache file is invalid (no regen for metadata change).
   // Delete the file and abort the placement.
   cm.deleteFile(placementMsg.stageId, placementMsg.cdsId,
                 placementMsg.getSiteId, path)
   return

// If another thread processed any of the following events while this thread
// wrote the cache file:
//    * template deployment
//    * clear cache
//    * record deployment
else if tpltMd.templateHash != cachedTplt.templateHash ||
     tpltMd.clearCacheId != cachedTplt.clearCacheId ||
     rcdState.getGuid != cachedTplt.getRecordStateGuid,

   // Then the cached file already expired.

   // If the instance can be regenerated,
   if cm.policySaysRegenerate(tpltMd, path),

      // Set its state to regen.
      iMd.setState(InstanceState.REGEN)
   else

      // Otherwise, delete the file & abort the placement.
      cm.deleteFile(placementMsg.stageId, placementMsg.cdsId,
                    placementMsg.getSiteId, path)
      return
```

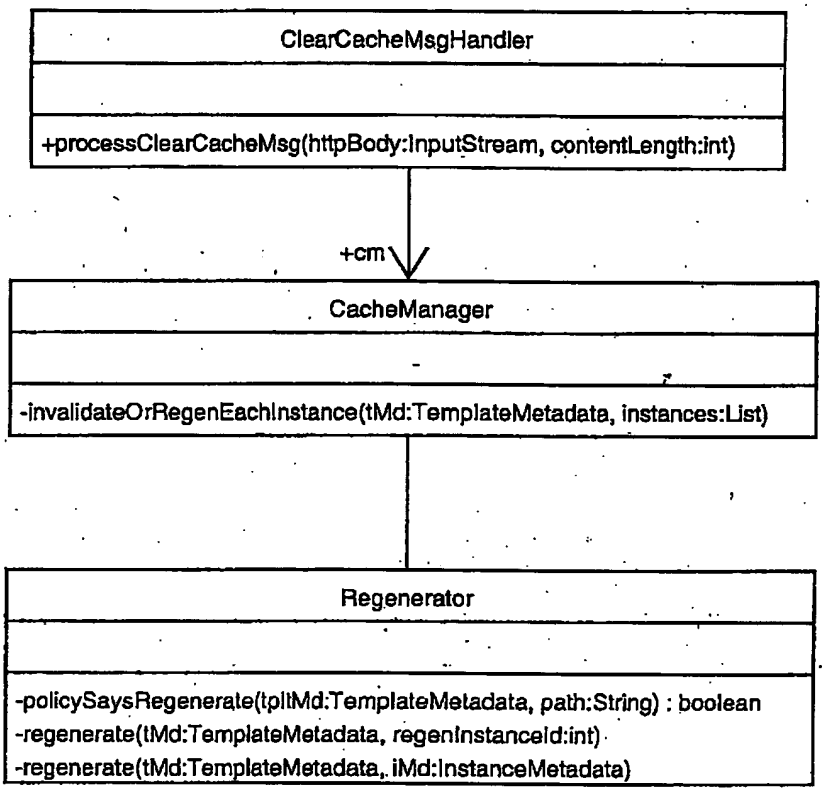
```
// Write inst and ws lookup metadata

iMd.setTemplateKey(tpltMd.getKey)
iMd.setParamHash(cachedTplt.getParamHash)
iMd.setVersion(version)
iMd.setRequestContentType(placementMsg.getResponseContentType)
iMd.setResponseContentType(placementMsg.getResponseContentType)
iMd.setRequestMethod(placementMsg.getRequestMethod)
iMd.setRegenHost(placementMsg.getHost)
iMd.setIsComponent(cachedTplt.getIsComponent)
iMd.setExtension(ext)

mdModuleValues = iMd.getModuleValues
for each plModuleValues in cachedTplt.enumerateModuleValues,
   set mdModuleValues from plModuleValues

List idList = iMd.getContentTrackingIds
for each id in cachedTplt.getContentTracking.getIds,
   idList.add(id)

List actionList = iMd.getActions
for each action in cachedTplt.getActions,
   Action actionMd = new Action
   actionMd.setHandler(action.getHandler)
   List actionMdParamList = actionMd.getParameters
   for each param in action.getParams,
      ActionParameter apMd = new ActionParameter
      apMd.setName(param.getName)
      apMd.setValue(param.getValue)
      actionMdParamList.add(apMd)

// If this is the outer cached page,
if cachedTplt.hasUrlAtWs,

   insertWebServerLookup(tpltMd, placementMsg, cachedTplt)
   iMd.setOuterUrlAtWs(cachedTplt.getUrlAtWs)

else // component or direct PG request

   iMd.setOuterUrlAtWs(outerUrlAtWs)

mdCon.insertInstance(iMd)

// End transaction
mdCon.commit
mdCon.setAutocommit(true)

// If another thread processed an event that caused the instance we just wrote
// to expire, and if the instance should be regenerated,
if iMd.getState == Instance.REGEN,

   // Send regenerate to WS
```

```
        cm.regenerate(tpltMd, iMd)


// Call with autocommit false
insertWebServerLookup(tMd : TemplateMetadata, placementMsg: PlacementMsg,
                      cachedTplt : CachedTemplate)

   // If another placement-msg concurrently updated the table, just quit
   if mdCon.hasWebServerLookup(cachedTplt.getUrlAtWs),
      return

   WebServerLookup wsLookup = new WebServerLookup
   wsLookup.setUrlAtWs(cachedTplt.getUrlAtWs)
   wsLookup.setTemplateKey(tMd.getKey)
   wsLookup.setStageId(placementMsg.getStageId)
   wsLookup.setCdsId(placementMsg.getCdsId)
   wsLookup.setSiteId(placementMsg.getSiteId)
   wsLookup.setPathAtPg(tMd.getPathAtPg)
   mdCon.insertWebServerLookup(wsLookup)
```

## 4.4 *ClearCacheMsgHandler*

Handles the PG-CM clear-cache-msg described on page 12.

The following diagram shows the classes that process the clear-cache-msg:



The following pseudo-code shows how the clear-cache-msg is handled:

```
processClearCacheMsg(httpBody : InputStream, contentLength : int)
   // Read the msg, then unmarshal it into a castor msg object
   byte[] buffer = new byte[contentLength]
   int bytesRead = httpBody.read(buffer)
   String string = new String(buffer, "UTF-8")
   StringReader stringReader = new StringReader(string)
   ccMsg = ClearCacheMsg.unmarshalClearCacheMsg(stringReader)

   String stageId = ccMsg.getStageId
   String cdsId = ccMsg.getCdsId
   String siteId = ccMsg.getSiteId

   // Get the templates that match the paths argument
   templates = mdCon.findTemplates(stageId, cdsId, siteId, ccMsg.getPaths,
         ccMsg.getClearCacheOption)

   for each tMd in templates,

      // Begin transaction
      mdCon.setAutocommit(false)

      tMd.setClearCacheId(ccMsg.getClearCacheId)
      tMd.update

      // Convert the castor match-map & match-list objects into the
      // parameters Map defined in the COD Clear Cache API.  This means
      // if a match-map and a match-list have the same type, wrap them
      // in a single list since you can't have 2 outer Map entries with
      // the same module name.

      // Get the valid instances to be cleared.  Skip regen instances, since
      // they've already been been cleared by some other event.  If PG's already
      // doing the regen, the placement clear-cache-id will control whether it's
      // placed or regen'd again.
      instances = mdCon.findInstances(tMd, ccMsg.getMatchMap, ccMsg.getMatchList)

      // End transaction
      mdCon.commit
      mdCon.setAutocommit(true)

      cm.invalidateOrRegenEachInstance(tMd, instances)
```

# 5 Cache Manager

The following diagram shows the CacheManager class data members and methods:

| CacheManager |
| --- |
| -host : String<br>-port : int<br>-cacheRoot : String<br>-cacheFileExtensionSsi : String<br>-cacheFileExtensionNoSsi : String<br>-componentSsiPrefix : String<br>-componentForwardPath : String<br>-modules : Map |
| -getFullFilename(stageId:String, cdsId:String, siteId:String, pathAtPg:String, paramHash:String, version:int, extension:String)<br>-getOutputStream(path:String) : OutputStream<br>-writeSsi(stream:OutputStream, outputOffset:int)<br>-writeSsi(stream:OutputStream, outputOffset:int, componentPathAtPg:String)<br>-closeOutputStream(stream:OutputStream)<br>-deleteFile(path:String)<br>-policySaysRegenerate(tMd:TemplateMetadata, iMd:InstanceMetadata) : boolean<br>-policySaysRegenerate(tMd:TemplateMetadata, path:String) : boolean<br>-regenerate(tMd:TemplateMetadata, regenInstanceId:int)<br>-regenerate(tMd:TemplateMetadata, iMd:InstanceMetadata)<br>-invalidateOrRegenEachInstance(tMd:TemplateMetadata, instances:List)<br>-invalidateOrRegenInstance(tMd:TemplateMetadata, iMd:InstanceMetadata)<br>-invalidateOrRegenEachInstance(instances:List) |

The following pseudo-code describes the CacheManager methods:

```
CacheManager.init(config : ConfigThingy)
   // Initialize my config variables
   config.get(cacheRoot, CACHE_ROOT)
   config.get(componentSsiPrefix, COMPONENT_SSI_PREFIX)
   cacheFileExtensionSsi = "shmtl"
   config.get(cacheFileExtensionSsi, CACHE_FILE_EXTENSION_SSI)
   cacheFileExtensionNoSsi = "vgn"
   config.get(cacheFileExtensionNoSsi, CACHE_FILE_EXTENSION_NO_SSI)

   Based on config info, load each module and invoke its init routine
   Store references to the modules in the modules Map data member


getFullFilename(stageId, cdsId, siteId, pathAtPg, paramHash, version, ext)
```

```
    return cacheRoot + "/" + stageId + "/" + cdsId + "/" + siteId + "/"
          + pathAtPg + "/" + paramHash + "_" + String.valueOf(version % 10) + "." +
          ext
```

```
getOutputStream(path : String) : OutputStream
```

Try to use java.nio.channels.FileChannel lock methods to only allow 1 thread to open the file at a time. Ideally, threads not getting the lock should return immediately or throw an exception. The placement would then be abandoned immediately. If java.nio can't do that either, then we may need to use JNI (see section 14.7.2.1 Problem 1 - concurrent open and write to the same file by different threads in the same process).

```
writeSsi(stream : OutputStream, offset : int) : int

    // Build the ssi
    String ssi = "<!--#" + componentSsiPrefix + "\"/" + "?vgnId=" + offset + "\"-->"

    // Convert it to bytes (should be ASCII, but utf-8 won't hurt)
    Byte[] bytes = ssi.getBytes("UTF-8")

    // Write the ssi as bytes, then return the # bytes written
    stream.write(bytes)
    return bytes.length

writeSsi(stream : OutputStream, offset : int, componentPathAtPg : String) : int

    // Build the ssi
    String ssi = "<!--#" + componentSsiPrefix + "\"" + componentPathAtPg
          + "?vgnId=" + offset + "\"-->"

    // Convert it to bytes (should be ASCII, but utf-8 doesn't hurt)
    Byte[] bytes = ssi.getBytes("UTF-8")

    // Write the ssi as bytes, then return the # bytes written
    stream.write(bytes)
    return bytes.length

closeOutputStream(stream : OutputStream)

deleteFile(path : String)

policySaysRegenerate(tMd : TemplateMetadata, iMd : InstanceMetadata) : boolean
    path = cm.getFullFilename(tMd.getStageId, tMd.getCdsId, tMd.getSiteId,
          tMd.getPathAtPg, iMd.getParamHash, iMd.getVersion, iMd.getExtension)
    return regenerator.policySaysRegenerate(tpltMd, path)
```

```
policySaysRegenerate(tpltMd : TemplateMetadata, path : String) : boolean
   return regenerator.policySaysRegenerate(tpltMd, path)


regenerate(tMd : TemplateMetadata, regenInstanceId : int)
   regenerator.regenerate(tMd, regenInstanceId)


regenerate(tMd : TemplateMetadata, iMd : InstanceMetadata)
   regenerator.regenerate(tMd, iMd)


invalidateOrRegenEachInstance(tMd : TemplateMetadata, instances : List)
   for each iMd in instances,
      invalidateOrRegenInstance(tMd, iMd)


invalidateOrRegenInstance(tMd : TemplateMetadata, iMd : InstanceMetadata)

   // Begin transaction
   mdCon.setAutocommit(false)

   // Set state to invalid or regen, depending on regenerate policy
   iMd.setState(policySaysRegenerate(tMd, iMd) ?
                InstanceState.REGEN : InstanceState.INVALID)

   // If the instance is transitioning to invalid, remember the expiration
   // timestamp for the instance deleter.
   if iMd.getState == InstanceState.INVALID,
      iMd.setExpirationTimestamp(new Date)

   // Else if the instance is transitioning to regen, send regenerate to WS
   else if iMd.getState == Instance.REGEN,
      regenerate(tMd, iMd)

   iMd.update

   // End transaction
   mdCon.commit
   mdCon.setAutocommit(true)


invalidateOrRegenEachInstance(instances : List)
   for each iMd in instances,
      tMd = mdCon.findTemplate(iMd)    invalidateOrRegenInstance(tMd, iMd)
```

# 6 Instance deleter

The instance deleter is a background thread that wakes up every CLEANUP_PERIOD seconds and deletes instances (instance metadata and cache file) whose state has been Invalid for more than CLEANUP_DELAY seconds.

Since CLEANUP_DELAY is intended to give clients time to receive the Invalid state change before the file is deleted, we shouldn't run into problems deleting the file because someone still has it open. So I'm not planning to do anything fancy for file delete. I'll just use java.io.File.delete. The file delete will occur before the instance metadata delete. If the file exists, and the delete fails, the instance metadata delete will be omitted, so the file delete will be tried again at the next cleanup.

# 7 Periodic expiration

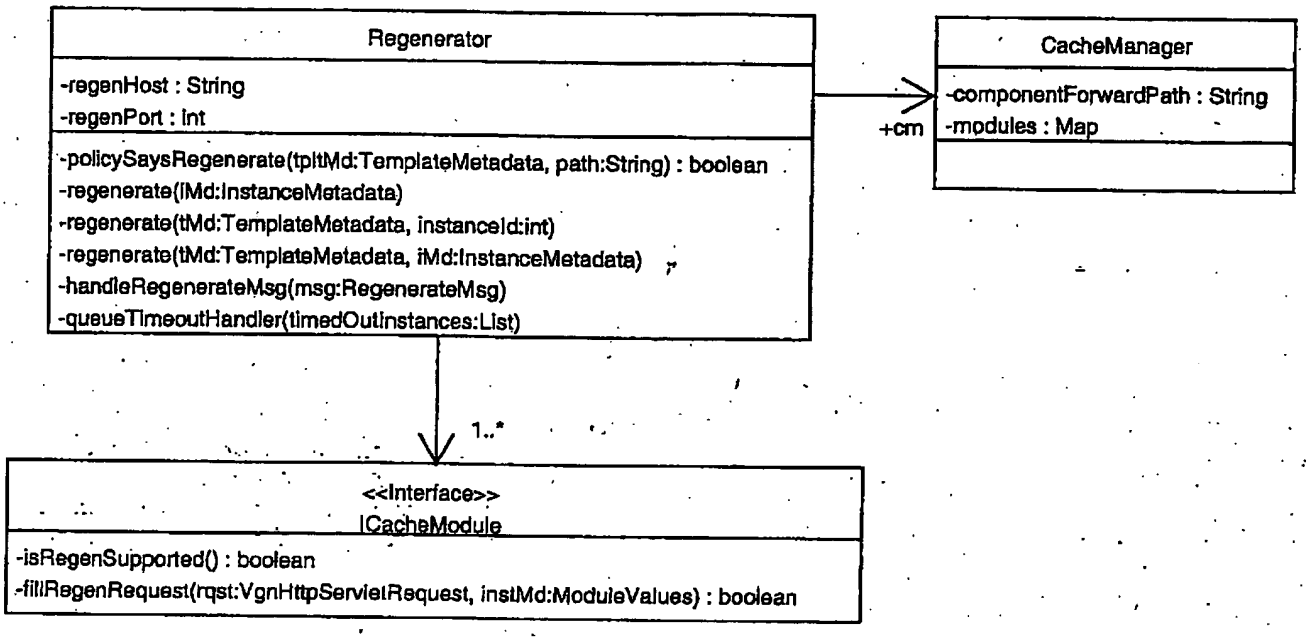
The periodic expiration thread expires instances according to their expire-policy and expire-period settings. When a template is added or an existing template's metadata changes, its template metadata is passed to periodic expiration so it can initialize expiration processing for the template if it's set to expire periodically, and if it's not already initialized. Periodic expiration keeps a min heap of the next expiration for each periodically expired template. The key to the heap is the next expiration time. The value is a list of templates whose next expiration time equals the key. When a template is deleted or an existing template's metadata changes, its template metadata is passed to periodic expiration so it can terminate expiration processing for the template if it's not set to expire periodically, and it's currently initialized. After all add/delete/update messages are processed, the thread sleeps till the next expire time or another message is received. When an expire time occurs, an entry is removed from the heap, and the following processing occurs:

```
for each tMd in the entry,
   instances = mdCon.getValidInstances(tMd)
   tMd.setPeriodicExpirationTimestamp(entry.getExpireTime)
   mdCon.update(tMd)
   cm.invalidateOrRegenEachInstance(tMd, instances)
   t - calculateNextExpireTime(tMd)
   addEntry(t, tMd)
```

# 8 Regenerator

A Regenerator object is also a thread. The REGENERATION_CONCURRENCY_LIMIT configuration variable is enforced by placing the specified number of Regenerators in the regenerator thread pool. Each Regenerator gets RegenerateMsgs from the thread pool input queue. The regenerate message and queue timeout handlers are described below.

The following diagram shows the classes that process regeneration:




```
Regenerator.init
   // Initialize my config variables
   config.get(regenHost, REGENERATE_HOST)
   config.get(regenPort, REGENERATE_PORT)

   TaskManager.setQueueTimeoutHandler(queueTimeoutHandler)

   instances = mdCon.findAllRegenInstances()
   for each iMd in instances,
      regenerate(iMd)
```

The following method is called by other classes to determine whether to regenerate:

```
policySaysRegenerate(tpltMd : TemplateMetadata, path : String)
   if tpltMd.regeneratePolicy == Never,
      return false
```

```
   if tpltMd.regeneratePolicy == Always,
      return true

   Date now = new Date
   Date lastAccess = FileIo.lastAccess(path)

   return (now.getTime - lastAccess.getTime) / MILLISEC_PER_SEC <=
         tpltMd.getRegenerateLastAccessedInterval
```

The 3 following methods are called by other classes to put a RegenerateMsg in the Regenerator thread pool's input queue:

```
regenerate(tMd : TemplateMetadata, instanceId : int)
   iMd = mdCon.getInstance(instanceId)
   regenerate(tMd, iMd)


regenerate(iMd : InstanceMetadata)
   tMd = mdCon.findTemplate(iMd.getTemplateKey)
   regenerate(tMd, iMd)


regenerate(tMd : TemplateMetadata, iMd : InstanceMetadata)
   puts a RegenerateMsg in the regenerator queue with timeout =
         iMd.expirationTimestamp + REGENERATE_MAX_TIME_TO_REQUEST


RegenerateMsg
   tMd : TemplateMetadata
   iMd : InstanceMetadata
```

The following method handles the RegenerateMsg:

```
handleRegenerateMsg(regenMsg : RegenerateMsg)
   tMd = regenMsg.tMd
   iMd = regenMsg.iMd

   // Let modules add their parameters to the request
   request = new VgnHttpServletRequest
   for each moduleValues in iMd.enumerateModuleValues,
      module = cm.modules.get(moduleValues.getModuleName)
      if ! module.isRegenSupported,
         return
      if ! module.fillRegenRequest(request, moduleValues)
         return

   // Set host & port
   if regenHost.length == 0,
```

```
    get host & port from iMd.getRegenHost
  else
    host = regenHost
    port = regenPort
  request.setServerName(host)
  request.setServerPort(port)

  // Set URI to outerUrlAtWs if there's a WS tier, or pathAtPg otherwise
  if iMd.hasOuterUrlAtWebserver,
    request.setURI(iMd.getOuterUrlAtWebserver)
    if iMd.isComponent,
      request.addHeader("x-vgn-component-path-at-pg", tMd.getPathAtPg)
      request.addHeader("x-vgn-stage-id", tMd.getStageId)
      request.addHeader("x-vgn-cds-id", tMd.getCdsId)
      request.addHeader("x-vgn-site-id", tMd.getSiteId)
  else
    request.setURI(tMd.getPathAtPg)

  if iMd.isComponent,

    // Add response-content-type header.  The filter sets the Content-type of
    // the response before invoking the component so the PG can transcode the
    // component output to match the encoding of the template that called the
    // component.
    request.addHeader("x-vgn-response-content-type", iMd.getResponseContentType)

  // Add Content-type header if a module didn't already do it.
  if request.getContentType == null && iMd.getRequestContentType != null,
    request.setContentType(iMd.getRequestContentType)

  // Add is-regen header.  This header tells the plug-in to not serve the request
  // from the cache.  The header is forwarded in the request to the PG.  PG sees
  // it and knows to not serve the request from the cache.
  request.addHeader("x-vgn-is-regen", "true")

  // Add identifiers of different events that can cause regenerations.  This helps
  // PG avoid redundant regenerations.
  request.addHeader("x-vgn-template-hash", tMd.getTemplateHash)
  request.addHeader("x-vgn-clear-cache-id", tMd.getClearCacheId)
  request.addHeader("x-vgn-periodic-expiration-timestamp",
      tMd.getPeriodicExpirationTimestamp)
  rcdState = mdCon.findRecordState(tMd.getStageId, tMd.getCdsId)
  request.addHeader("x-vgn-record-state-guid", rcdState.getGuid)

  // Add param-hash header so filter doesn't have to compute it unnecessarily.
  request.addHeader("x-vgn-param-hash", iMd.getParamHash)

  send request
  wait for the response with timeout = REGENERATE_TIMEOUT
  if response received && response.isOk,
    placementMsg = response.getPlacementMsg

    // PlacementMsgHandler checks regenInstanceId in places where regen placement
```

```
      // differs from normal placement.  These cases are caused by race conditions
      // between different types of expiration events on the same instance.  The
      // races conditions only occur when the template's regenerate-policy causes
      // regeneration.  For example, if a clear-cache occurs for an instance that's
      // already being regenerated due to a template update, the regen may or may
      // not pick up the new clear-cache-id.  If the placement handler sees that a
      // regen placement's clear-cache-id is old, it restarts the regeneration,
      // whereas an old normal placement is just discarded.
      placementMsg.setRegenInstanceId(iMd.getKey)

      send placementMsg to PlacementMsgHandler TaskManager
   else if timeout or response failure == OldTemplate,
      // Timeout or failure - invalidate the instance.
      iMd.setState(Invalid)
      iMd.setExpirationTimestamp(new Date)
      mdCon.updateInstance(iMd)

   // Else failure was OldTemplate.  That means the PG that received the regenerate
   // did not have the version of the template that caused the request.  Leave the
   // instance as-is, because a subsequent template-update will be received from
   // the AM of the PG that returned OldTemplate, after which requests to that PG
   // will succeed.  So worst case, the request caused by the last AM's template-
   // update is guaranteed that all PGs have the new version of the template.


// Handles RegenerateMsgs that time out in the Regenerator queue.
queueTimeoutHandler(timedOutInstances : List)   for each iMd in timedOutInstances,
      iMd.setState(Invalid)
      iMd.update
```

## *8.1 Regenerate request*

This is an HTTP request sent to the host:port from the client request that caused the original instance to be cached.

Websphere is the lowest common denominator. Its Servlet Engine (SE) talks OSE rather than HTTP, so CM can't easily send regenerate requests directly to a Websphere SE. The solution is to save in the instance metadata the Host header from the client request that caused the original instance to be cached. Regenerations are then sent to the saved host:port. The REGENERATE_HOST and REGENERATE_PORT configuration variables override this behavior by specifying a host:port to which all regenerate requests are sent.

# 9 Configuration Space

## *9.1 Variables at the CM/cache level*

### 9.1.1 Public variables

#### 9.1.1.1 HOST

The name of the machine CM executes on.

#### 9.1.1.2 PORT

Type - integer

The port CM listens to for HTTP messages from AM and PG.

Min - 1

Max - 65535

#### 9.1.1.3 CACHE_ROOT

Type - String

The fully qualified path of the cache's directory root.

#### 9.1.1.4 CACHE_FILE_EXTENSION_SSI

If a cacheable template calls any components, its cache file is saved with this extension. Server-side includes must be enabled at the web server for the specified extension.

If a cached file does not call any components, it will be saved with the extension from CACHE_FILE_EXTENSION_NO_SSI.

Default - shtml (IIS, Apache, and iPlanet automatically do server-side parsing of .shtml files).

Dynamic change - start using the new value. Customer needs to follow the following procedure when changing this variable:

1. Configure your web server to parse the new extension
2. Change CACHE_FILE_EXTENSION_SSI to the new extension
3. Execute VgnCDS.clearCache(new String[] {"/*"}, null)
4. Configure your web server to **not** parse the **old** extension

#### 9.1.1.5 CACHE_FILE_EXTENSION_NO_SSI

If a cacheable template does not call any components, its cache file will be saved with this extension. Server-side includes should **not** be enabled at the web server for the specified extension.

If a cached file calls any components, it will be saved with the extension from CACHE_FILE_EXTENSION_SSI.

Default - vgn.

Dynamic change - start using the new value.

#### 9.1.1.6 COMPONENT_SSI_PREFIX

This config var will go away for one of the following reasons:

1. exec CGI is probably an unacceptable security hole. Jam is researching whether we can make include virtual work for IIS.
2. Even if exec CGI turns out to be ok, the prefix should be derived from the TYPE {IIS, iPlanet, Apache, etc} of the WSs associated with the cache.

*When a cacheable template calls a component, the component output is extracted from the template's output and replaced with a server-side include (SSI) that either retrieves a cached component or sends a component request to the PG. The type of SSI directive depends on the web server type.*

*COMPONENT_SSI_PREFIX contains the string between "<!--" and "=" in the SSI directive.*

*For IIS web servers, the default is "exec CGI".*

*For other web servers, the default is "include virtual".*

#### 9.1.1.7 POOL_SIZE

Type - integer

Placements, template changes, record changes, and clear cache commands are all handled by the same thread pool. This configuration variable determines how many threads are in the pool.

Min - 1

Max - 256

Default - 10

### 9.1.2 Hidden variables

These variables control internal workings that the customer probably doesn't need to worry about, but

we don't want to hard-code. We would like to handle them something like the config variables in V6 that are added to template files in comments. The variables are not automatically created, and they are not documented. But if created, code will use them. This allows support or VPS to alter the behavior if necessary, but hides unnecessary complexity from the customer.

#### 9.1.2.1 CLEANUP_PERIOD

Type - integer

The instance deleter wakes up every CLEANUP_PERIOD seconds to delete invalid instances.

Min - 0

Max - 2592000 (30 days)

Default - 600

#### 9.1.2.2 CLEANUP_DELAY

Type - integer

When the instance deleter wakes up, it only deletes instances (instance metadata and cache file) whose states have been Invalid for more than CLEANUP_DELAY seconds. The delay makes sure the instance metadata has time to propagate to all clients before the cache file is deleted. Although the propagation should occur within a second, the default is conservatively set a couple of orders of magnitude larger.

Min - 0

Max - 2592000 (30 days)

Default - 300

## *9.2 Variables at the stage:CDS or stage:CDS:site level*

For maximum granularity, a separate set of these variables can be defined at each intersection between CDS and site. Or they can be defined at the CDS level if the values are the same for all the sites in a CDS. If a variable is specified for a CDS and one of the sites associated with that CDS, the CDS:site value overrides the CDS value.

### 9.2.1 REGENERATE_HOST

The name of the machine CM sends regenerate requests to. If blank, CM sends each regenerate request to the host:port that received the request that cached the instance.

### 9.2.2 REGENERATE_PORT

Type - integer

The port CM sends regenerate requests to. Used when REGENERATE_HOST is not blank.

Min - 1

Max - 65535

### 9.2.3 REGENERATE_CONCURRENCY_LIMIT

Type - integer

The maximum number of concurrent regenerate requests a CM makes to a Page Generator (PG) when multiple instances need to be regenerated.

Min - 1

Max - 1024

Default - 2

### 9.2.4 REGENERATE_MAX_TIME_TO_REQUEST

Type - integer

The maximum number of seconds from expiration that an old instance resides in the cache while its regenerate request waits to be issued to a PG. If the regenerate request can't be sent within this time, the old instance is invalidated, and the regenerate request is discarded.

Min - 1

Max - 3600

Default - 300

### 9.2.5 REGENERATE_TIMEOUT

Type - integer

The maximum number of seconds CM waits for the regenerate response to a regenerate request. If the timeout occurs, the old instance is invalidated.

Min - 1

Max - 3600

Default - 60

# 10 Policy definition file

expire-policy and regenerate-policy were intentionally represented as enumerations (rather than distinct booleans like ContentChange) to simplify selection of the useful business cases.

## *10.1 expire-policy*

One of the 4 following values:

1. ContentChange - yields freshest pages. Template must use content tracking API.
2. PeriodicIfContentChanges - use if content changes too fast for policy 1. Template must use content tracking API.
3. Periodic - use if (template does not use content tracking API or uses content not managed by VCM) and (content changes on a regular basis).
4. None - the default. Instance expires when template is updated or cleared via clear cache API command. Use if (template does not use content tracking API or uses content not managed by VCM) and (content changes under control of another template). The template that initiates the content change must use clear cache API to expire templates that use the new content.

## *10.2 expire-period*

**still need to think about:**

- **illegal combinations like day-of-month=29 or 30 for February**
- **daylight savings time changes**

expire-period is specified in the following sub-elements: minute, hour, day-of-month, month-of-year, and day-of-week. Each sub-element is a string conforming to the following syntax:

* | elementList | everyXUnits

* = all legal values

elementList = element, element, ...

element = number | inclusiveRange

inclusiveRange = number-number

everyXUnits = /number

Days may be specified 2 ways: day of the month and day of the week. If both are specified, they are

additive.

The time zone is the local time zone of the machine running cache manager.

Examples:

| minute | hour | day-of-month | month-of-year | day-of-week | result |
|---|---|---|---|---|---|
| 0 | 8 | * | * | * | every day at 8:00 am |
| 0, 30 | * | * | * | * | every 30 minutes |
| 0 | 0 | 1, 15 | * | 1 | midnight every 1st and 15th and every Monday |
| /2 | * | * | * | 1-5 | every 2 minutes Monday-Friday |

**10.2.1 minute (0-59)**

Minutes of the hour that the template expires on

Legal values are 0-59

Default - 0

**10.2.2 hour (0-23)**

Hours of the day that the template expires on

Legal values are 0-23

Default - 0

**10.2.3 day-of-month (1-31)**

Days of the month that the template expires on

Legal values are 1-31

Default - *

**10.2.4 month-of-year (1-12)**

Months of the year that the template expires on

Legal values are 1-12

Default - *

#### 10.2.5 day-of-week (0-6 with 0=Sunday)

Days of the week that the template expires on

Legal values are 0-6 with 0=Sunday

Default - *

### *10.3 regenerate-policy*

One of the 3 following values:

1. Never
2. IfLastAccessedWithinRegenerateLastAccessedInterval
3. Always

### *10.4 regenerate-last-accessed-interval*

Type - number of seconds

The interval following the template instance's last access time during which expiration causes regeneration to occur.

The granularity depends on the web server type. See the Webserver Plugin design document [11] for more information.

# 11 Plug into generic framework

The cache manager runs as an aglet in the vgnagent framework. vgnagent is a Vignette managed process from config's point of view. For cache manager, COD config sets up a vgnagent with 2 aglets: HttpAglet and CmAglet. The ordinal of CmAglet must be less than that of HttpAglet. In addition, config must add the following common.http.action mappings for CmAglet:

- /placement - com.vignette.cod.cm.PlacementMsgReceiver
- /metadata - com.vignette.cod.cm.MetadataMsgReceiver
- /recordUpdate - com.vignette.cod.cm.RecordUpdateMsgReceiver
- /clearCache - com.vignette.cod.cm.ClearCacheMsgReceiver

vgnagent calls initialize and start methods on each aglet in ordinal order. It calls the shutdown method on each aglet in reverse ordinal order.

CmAglet.initialize initializes the singleton CacheManager object. Then when HttpAglet initialize occurs, and the cm message receivers are initialized, then can get what they need from the CacheManager singleton.

## *11.1 Message receivers*

The CM message receivers extend HttpAction and are invoked by the HttpAction infrastructure with a reference to an HTTP message. Each receiver invokes its corresponding message handler.

# 12 Monitoring attributes

Here's a 1st cut at the metrics that CM can collect. I also list some collected by WS and PG, because they seem basic to seeing how caching is performing. And some listed as collected by CM could be collected by PG during page generation instead.

Another idea which might replace some of the metrics below like # regens timed out waiting to be sent to PG is to have infrastructure elements like Queue instrument the current # items in the queue.

- CM
  - # normal placements
  - # regen placements
  - # instance expirations caused by clear cache
  - # instance expirations caused by template update
  - # instance expirations caused by record update
  - # instance expirations caused by periodic expiration
  - # failed regenerations
  - # regenerations that time out waiting to be sent to PG
  - # regenerations that time out waiting on PG response
  - list of active metadata clients
  - list of disconnected metadata clients (perhaps)
- PG
  - time to generate
  - # cache hits
- WS plugin
  - # cache hits

# 13 Open issues

## *13.1 Vicket 59722(CUST: Page regeneration has probably never worked correctly on Windows NT/2000)*

This is about getting cached file last access time via Win32 FindFirstFile vs. GetFileInformationByHandle. The former can return a stale value. What does java use? Try to reproduce the problem with COD.

## *13.2 Vicket 8233(CUST: CMD should delete directories when template is expired/deleted)*

When metadata-msg.template-entry.template-event is delete, make sure to delete the cacheRoot/pathAtPg subdirectory after deleting all the instances.

## *13.3 Clearing WebserverLookup table when WS path to PG path mapping changes*

Do we add a new API command? Probably not since it should be extremely rare.

Does CM listen for an HTTP msg sent to placement port by some standalone utility? From email from Isaac:

> It may be better if we provide an option to the metadata tool to clear this table or perhaps clear for specific web server path patterns. Then the question becomes - how does this get refilled? I guess it should happen as a result of normal placement because the web server will not find the cached page. A subsequent placement will be for an existing cached page - but, the CM should always try to update the mapping table even if the placement can be ignored...

Rather than learning the WS-PG path mapping, should we just add an XML element in the CDF or PDF to specify ws-path-pattern to pg-path-pattern mapping?

## *13.4 Exploded deployment and CDF external to .war file deployment order problem*

If a developer makes changes to the CDF and a template to ***add*** sensitivity parameters, and if the template changes are deployed before the CDF changes, there exists a potentially significant timing window where requests containing one of the new parameters will produce output dependent on the new parameter, but the instance will be cached without the new parameter. So subsequent requests without the new parameter will return results that are only produced when the new parameter is present.

The solution is to make sure the **CDF changes** are deployed *before or with* the **template changes.**

If a developer makes changes to the CDF and a template to ***remove*** sensitivity parameters, and if the CDF changes are deployed before the template changes, there exists a potentially significant timing

window where requests containing one of the obsolete parameters will produce output dependent on the obsolete parameter, but the instance will be cached without the obsolete parameter. So subsequent requests without the obsolete parameter will return results that are only produced when the obsolete parameter is present.

The solution is to make sure the **template changes** are deployed *before or with* the **CDF changes**.

If a developer adds sensitivity parameters to 1 template and removes parameters from another template and both templates belong to the same webapp (and therefore use the same CDF), then the solutions for the 2 previous bullets are incompatible.

The solution for all 3 cases is to make sure the CDF changes are deployed ***with*** the template changes. This requires vcm support for grouping changes to a subset of content items within a deployment group into a single deployment. This functionality has use beyond COD, especially for exploded deployment. It could work like Perforce's changelist functionality.

### *13.5 Too many cache files in a directory*

Miguel's testing on this issue may not be complete. So far, the results suggest that our current scheme of just 1 directory per template may be fine. However, we will continue testing to see if it becomes a problem.

### *13.6 Configuration bootstrapping*

What is the sequence for configuring COD components? We need to figure out the basic scenarios.

### *13.7 Configuration teardown*

What is the sequence for tearing down COD components? We need to figure out the basic scenarios.

### *13.8 Failover*

How will we do failover? It could be as simple as configuring a backup CM and sleepycat that can be started by 3rd party failover software. The backup CM and sleepycat then take control of the repository and cache over from the primary.

# 14 Closed Issues

## *14.1 Preventing a cached component in a cached template from being served directly from the WS cache*

Web.xml can map a URI pattern like /component/* to an error page. Templates with these paths can then only be accessed via include or component commands. If one of these components is cached and is also called by a cached template, then both the template output and the component output will be cached in the WS cache. If a browser directly requests the component, our plugin should not serve the request from the cache.

The solution is:

- If the template is invoked via the component command, a string like "component" will be added to the request data to be hashed into the filename.
- When handling SSIs, the plugin will add "component" to the request data to be hashed, thus matching the filename in the previous step.
- When handling direct requests, the plugin will *not* add "component" to the request data to be hashed, thus missing the component in the WS cache.

If a component is not prevented from direct access by web.xml uri mapping, and if it's requested both directly and as a component, then 2 instances will go in the cache, 1 with "component" in the filename & the other without.

## *14.2 Template output differs based on whether invoked directly or as a component*

If a template's behavior differs based on whether it's invoked directly or as a component, then the template must contain logic like this:

```
if ( request.getRequestURI() == "/myURI.jsp" )
   out.println( "one thing" );
else
   out.println( "another" );
```

Do the following things to make sure output is properly cached:

- Use web.xml to map 2 URIs to the template - 1 for direct access and another for component access.
- Add an entry for each URI to the CDF.
- Use the component URI in component command (as opposed to the unmapped URI).
- Use the component URI in comparisons with HttpServletRequest.getRequestURI.

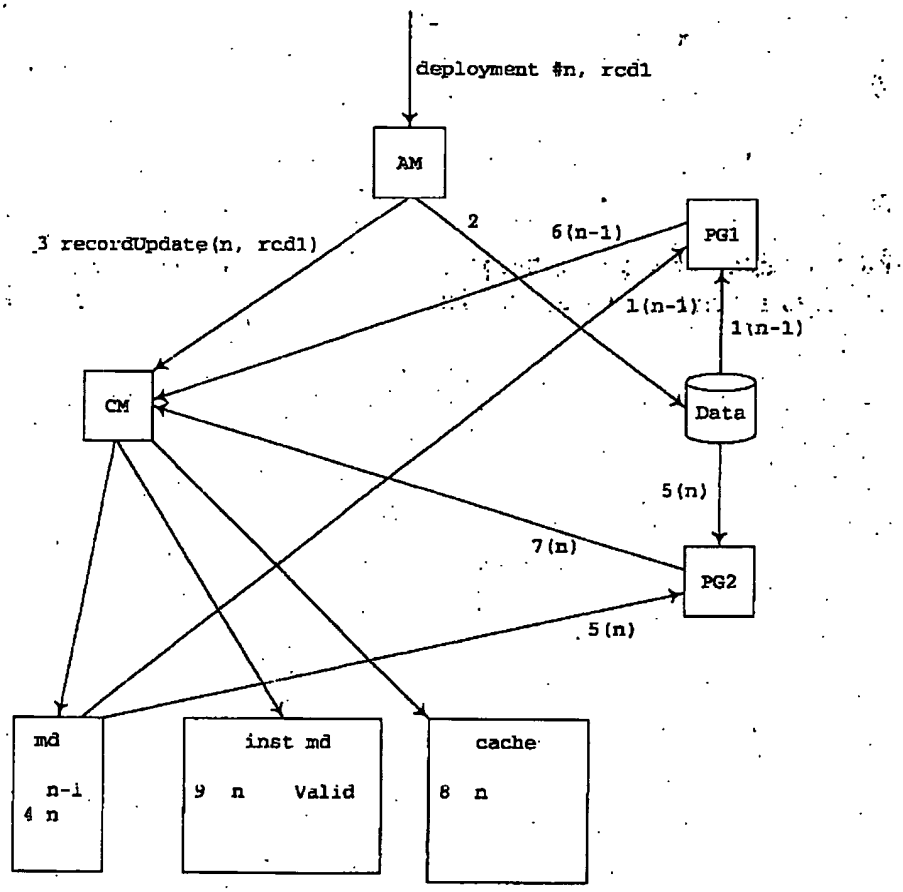
This also allows the direct and component instances to have different caching parameters.

### *14.3 Vicket 57778(page regeneration doesn't do what it says on the tin..)*

iPlanet & IIS apparently cache pages in memory. When they serve from memory, our disk file's last access time doesn't get updated. For now, the solution is to explain the situation so the customer can work around the problem. See section 10.4 regenerate-last-accessed-interval for my attempt to do so.

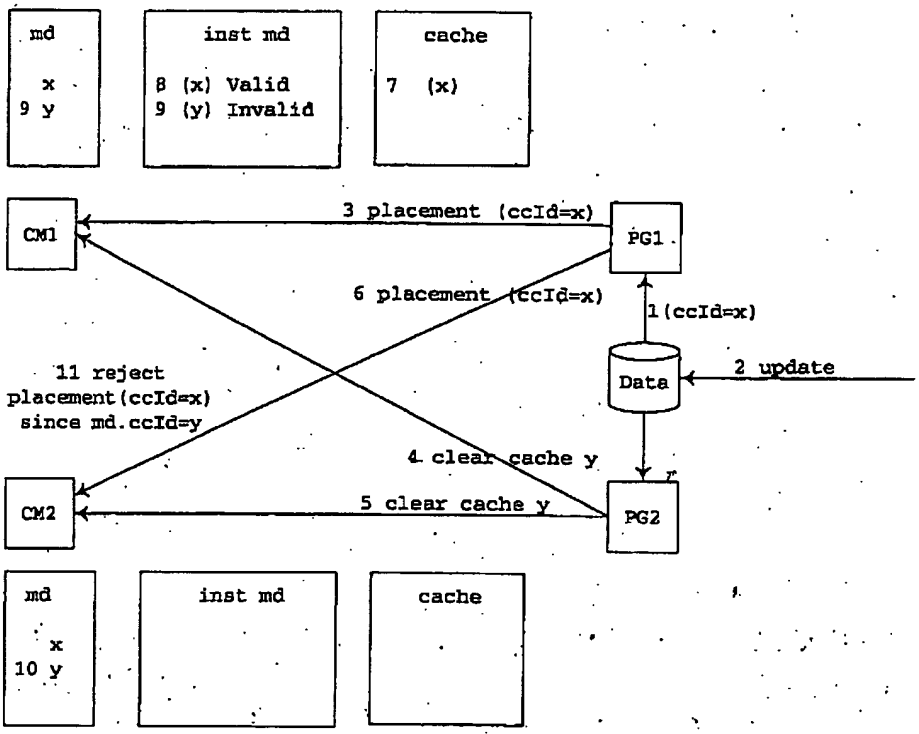
### *14.4 Record state guid*

The following scenario shows the record state guid being used to avoid caching a stale file. A page generation using old content completes (event 1) just before new content arrives (event 2). But CM processes the record update (events 3-4) before the placement with old content (event 6). CM then ignores the old placement (event 6) since its record state guid is less than the metadata record state guid. A page generation that occurs after the record update (event 5) yields a placement (event 7) that is later accepted (events 8-9).



## *14.5 Clear cache ID*

The following scenario shows how PG generates a unique clear cache ID to handle a race condition between placement and clear cache.




A template executes on PG1 and grabs old data. The data is updated, then clear cache is executed on PG2. PG1 broadcasts the placement built from old data. PG2 broadcasts the clear cache. CM1 receives and processes the placement before the clear cache. CM2 receives and processes the clear cache before the placement. CM2 rejects the placement with ccId=x since it doesn't match md.ccId=y.

## *14.6 Instance version*

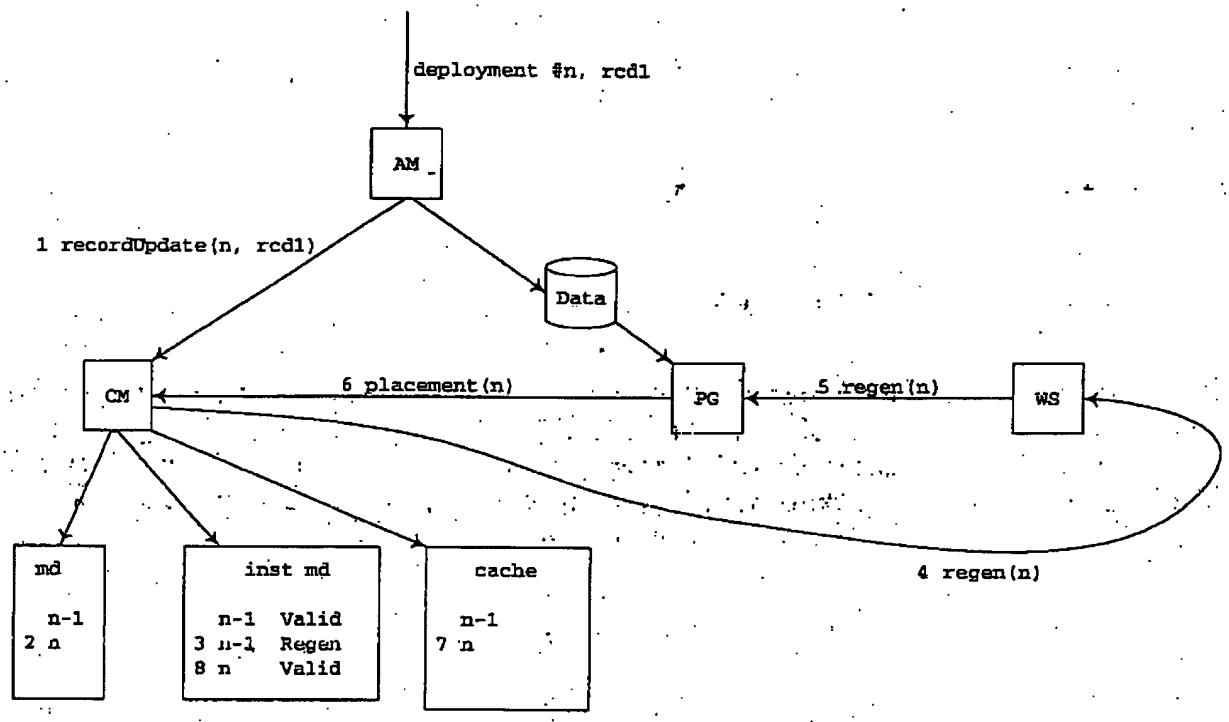
The next 2 scenarios show cases where a version mismatch between the cache file and the instance metadata record occurs unless multiple instance versions are allowed. In each case, there is a window in which WS or PG can serve a cache file generated from content version n with instance metadata produced by content version n-1. The sensitivity parameters in the instance metadata will be the same for both versions (otherwise it would be a different instance), but non-sensitivity instance data like content tracking IDs and RMS events can be different for same instance when generated with different versions of content.

The solution is to temporarily keep both the old and new version of the instance metadata and cache file. The instance version will be included in the cache filename not by including it in the parameter hash, but by appending it modulo 10 to the parameter hash (e.g., /cacheRoot/pathAtPg/paramHash_v.shtml where

v = version % 10). A WS or PG metadata query for an instance can return multiple rows (probably 2 at most). The latest version will be used when serving the request. The version is a counter incremented during placement. The old version will be deleted long enough after writing the new instance metadata to be reasonably confident the instance metadata has reached all in memory database replicas (the default timeout is several orders of magnitude larger than the normal propagation delay). The timeout is controlled by the CLEANUP_DELAY config variable described on page 45.

### 14.6.1 Regen placement overwrites file

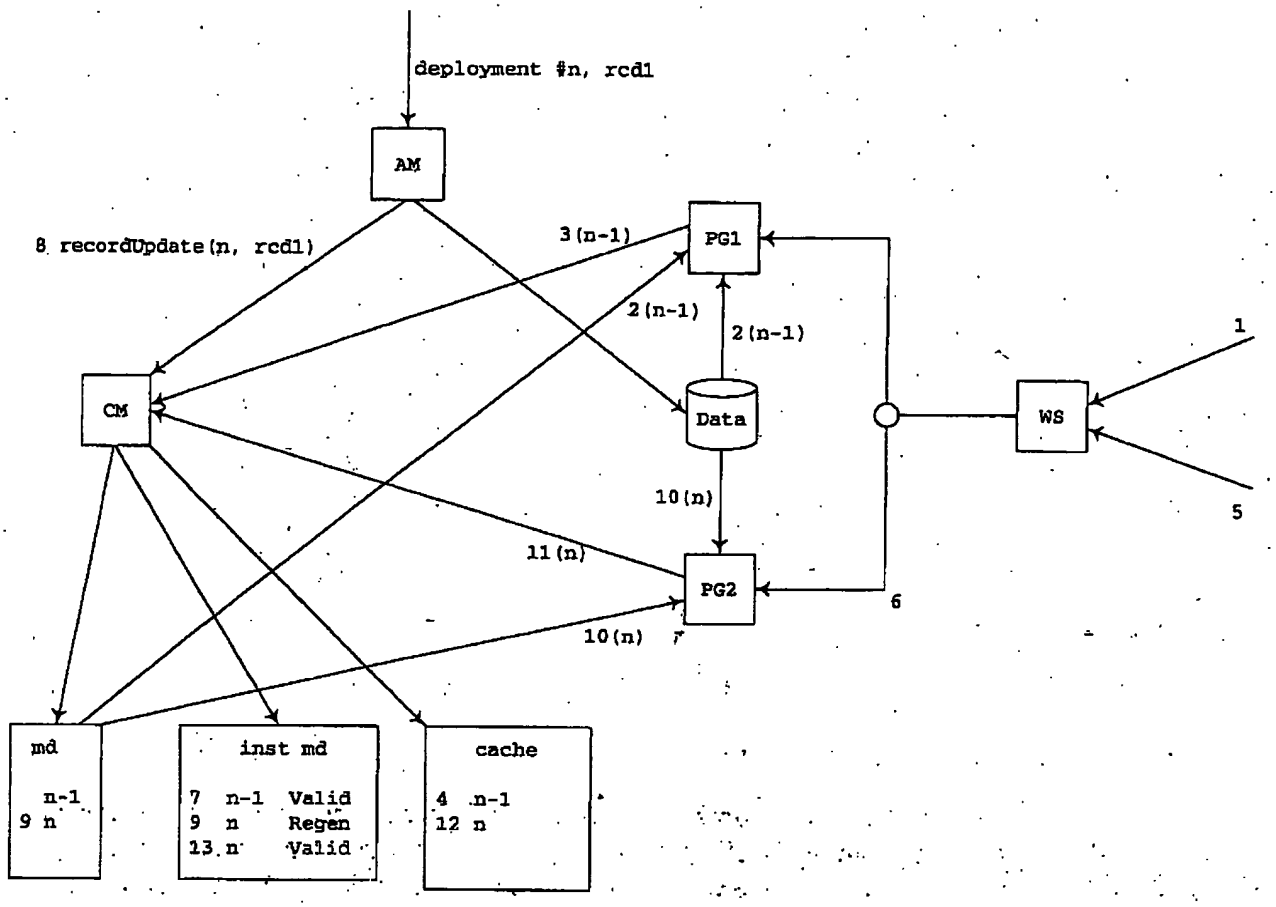
The mismatch occurs in a window that starts when the cache file is written with data from content deployment n (event 7) and ends after the instance metadata is written *and* propagated to the WS and PG (event 8 + replica propagation).




### 14.6.2 Normal placement overwrites file

The mismatch occurs in a window that starts when the cache file is written with data from content deployment n (event 12) and ends after the instance metadata is written *and* propagated to the WS and PG (event 13 + replica propagation).

The window is only open when regeneration occurs, because WS and PG serve from the cache when an instance is in the Regen state. Without regen, processing of the record update (event 9) would have set the instance state to Invalid. Invalid would probably propagate to WS and PG well before the new version of the cache file is written (event 12), so there should not be a problem in the no regen case.



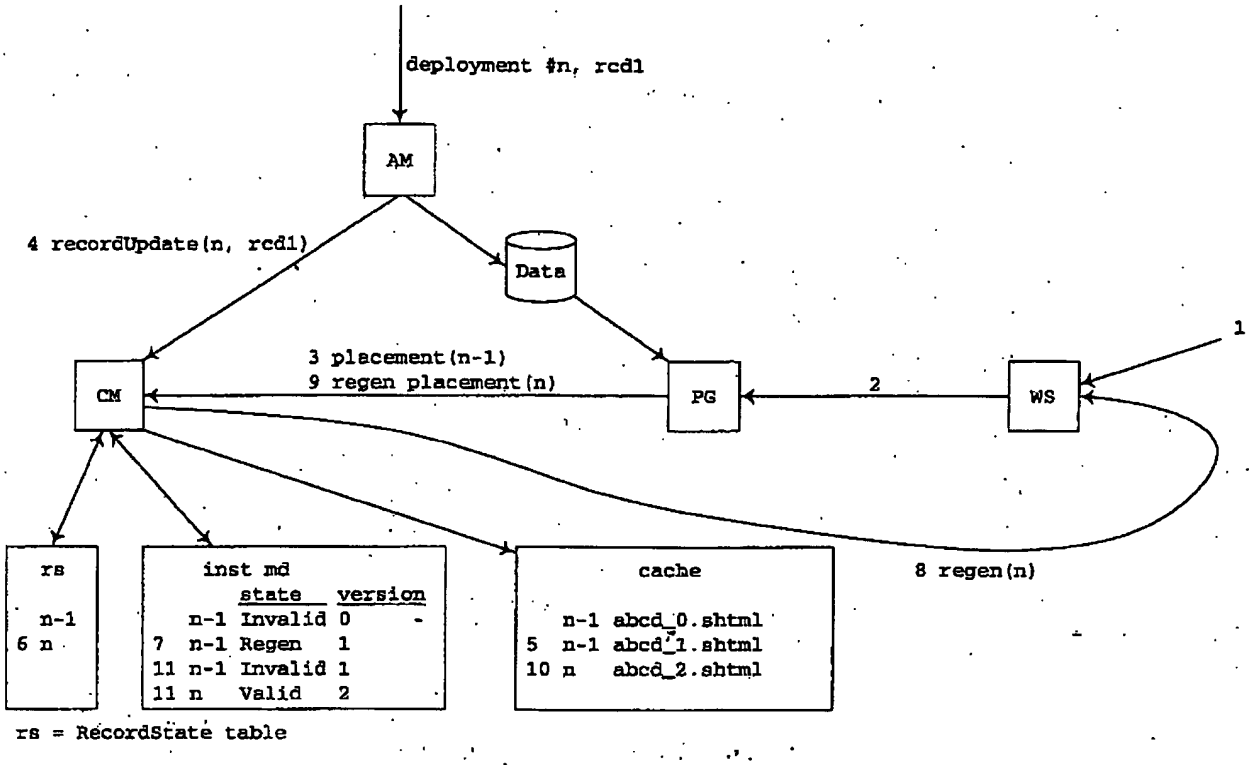

The diagram does not show the regeneration request corresponding to the instance state transition to Regen (event 9). That regen request is still in progress when the 2nd normal placement makes a preemptive strike to transition the instance state from Regen to Valid (event 13). When the regen placement arrives later, it's ignored, since a Valid instance already exists.

## *14.7 Metadata record locking*

The 2 following scenarios show cases where concurrency **within** CM requires in-memory DB record locking to ensure correct behavior.

### 14.7.1 Record update after CM starts writing cache file

The following scenario shows how in-memory database locking handles a race condition where a record update tries to update the RecordState at the same time an old placement tries to update the instance metadata table. In this case, the RecordState update occurs first (event 6). The instance metadata update (event 7) detects the RecordState update and sets the instance state to Regen (and sends a regen request) rather than taking the normal action of simply setting the state to Valid.




The locking is controlled by the JDBC Connection isolation level. We probably want SERIALIZABLE. From the TimesTen developer's guide release 4.3:

> When running at SERIALIZABLE transaction isolation level, TimesTen holds all locks for the duration of the transaction, so:
> • Any transaction updating a row blocks out readers and writers until the transaction commits.
> • Any transaction reading a row blocks out writers until the transaction commits.

In this scenario, the following transactions compete, and record update wins:

record update (event 6):

```
RecordState rcdState = mdCon.getRecordState(rcdUpdt.getStageId, rcdUpdt.getCdsId)
rcdState.setGuid(recordUpdateMsg.getGuid)
rcdState.write
mdCon.commit
```

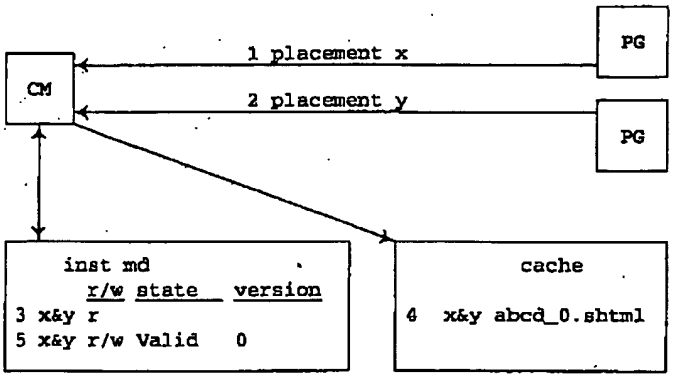
write instance metadata (event 7):

```
RecordState rcdState = mdCon.getRecordState(placement.getStageId,
      placement.getCdsId)
if  rcdState.getGuid == placement.getRecordStateGuid
   instMetadata.state = Valid
else
   if instance should be regenerated,
      instMetadata.state = Regen
   else
      instMetadata.state = Invalid
... set other instance metadata fields ...
instMetadata.newRecord
mdCon.commit
if instMetadata.state == Regen,
   send regen to WS
else if instMetadata.state == Invalid,
   delete cache file
```

Since the update occurred 1st, the instance metadata write sees the record state mismatch, and transitions to either Regen or Invalid. If the instance metadata write had occurred 1st, the instance would have been written in the Valid state, then the subsequent record update would have either invalidated it or started its regeneration.

### 14.7.2 Concurrent placements

The previous scenario showed that the metadata DB transaction that writes instance metadata must verify the template metadata record state guid still matches after the cache file is written. The same is true of the other synchronization mechanisms (templateHash, metadataHash, and clearCacheId). The following scenario shows that a similar thing can happen with the version, but the solution is different.




CM processes placements x and y concurrently on 2 different threads. Both threads read the instance metadata and see that there is no record (event 3), so both decide to write version 0 of the cache file (event 4).

#### 14.7.2.1 Problem 1 - concurrent open and write to the same file by different threads in the same process

Here are 2 basic options for solving the concurrent file access problem:

1. Use file locking to serialize file access
2. Use some other mutual exclusion mechanism to serialize file access

#1 seems simplest and cleanest. It would be best if this could be done with java file access classes, but my initial experiments with java.io.FileOutputStream don't show the locking behavior or level of control we need to serialize the accesses. However, java.nio.channels.FileChannel has lock methods that may do the trick. I haven't experimented with them yet. If we can't count on java 1.4, or if the 1.4 java.nio classes still don't cut the mustard, I guess we'll have to use JNI.

Initially, I planned to have the 2nd thread that tries to open the file block till the 1st thread closed the file, then the 2nd thread's open would succeed. It would actually be better if the 2nd thread's open returned immediately or threw a lock exception, so the 2nd thread could give up on the redundant placement immediately. That avoids problem 2 below.

If immediate lock detection is not possible, the threads not getting the lock should block until closeOutputStream is called. Then see Problem 2 below.

#### 14.7.2.2 Problem 2 - concurrent insert of same row into instance metadata

If the concurrent file access problem is solved so that the 2nd thread simply overwrites the file written by the 1st thread. Then both threads try to create a new instance metadata row. That's the next problem - the 2nd thread's insert will fail. Here are 2 basic options for solving the problem:

1. If the failure is "duplicate record," just ignore it.

2. Check the version in the transaction doing the new record, and roll back if it changed since it was read to calculate the cache file version.

I'm going with the "ignore the error" solution, because it doesn't require an extra query for the normal case when this race condition doesn't occur.

#### 14.7.2.3 Adding a new state

Another option for solving the concurrent placement problem is to add a Placing state to the instance state diagram. Before writing the cache file, the placementMsgHandler writes the instance metadata with a state of Placing in the same transaction that queries for the max version #. Another thread trying to place the same instance sees the new instance in the Placing state and realizes it is a redundant placement and abandons it.

The "file locking" solution is simpler, and it's definitely better as long as the locked file can be detected immediately. Immediately abandoning the placement frees the 2nd thread for other work.

If the lock can't be immediately detected, then I'd probably still go with the file locking solution for simplicity's sake, but that introduces the performance penalty of duplicate file writes, and the 2nd thread also loses the time spent blocked waiting for the 1st thread's write to complete. If those things cause performance problems in the real world, we could always switch to the "add a new state" solution.

## *14.8 File locking*

### 14.8.1 V6 vs COD file handling

- V6
  - Expire without regen: **rename** curl.html -> curl.html.bak
    - NT - uses a quite complicated method MTmv:
      - tries MoveFileExW; returns if ok
      - open src file & get a file mapping; share delete, read, & write
      - open dst file w/truncate, share delete, read, & write
      - if dst file not found, open w/create new, share delete, read & write
      - for each chunk of src, MapViewOfFile, WriteFile chunk to dst
      - close src file & mapping
      - open existing src file, delete on close, share delete, read, & write
      - close src
    - Other - VU_OS::rename
  - Expire with regen: **rename** temp.html => curl.html
    - Calls callPageDaemon in clients/common/generatePageImpl.cpp w/flag that WS writes cached pages
      - VU_OS_rename(tmpFile, curlFile, removeSrc=T)
        - NT - CopyFileW(tmpFile, curlFile, failIfExists=F); wunlink(tmpFile)
        - Other - rename
      - VU_OS_unlink(curl.html.bak)
        - NT - _wunlink
        - Other - unlink
- COD
  - Normal placement:
    - Multiple threads can concurrently try to open & write the same file. Another process will not have the file open, because it's a new version. cm.create/closeOutputStream will try to use java.nio file locking when creating the file s.t. 2nd thread is blocked until stream is closed. 2nd thread's open then succeeds & 2nd create truncates the file & rewrites it.
  - Regen placement:
    - Create new version of file. Don't worry about multiple threads creating at once, since there will only be 1 regen placement
  - Expire without regen:
    - Instance deleter thread **deletes** file. java.io.File.delete fails if dst is open by another jvm, but it should still do since deleter doesn't delete instances till enough time has passed to be assured no one is still using the instance.

**Resolution:** instance version, instance state, and the instance cleanup thread collaborate to avoid the file locking problems encountered by V6.

### 14.8.2 Vicket 12365(IIS file locking interferes w/ PAD, CMD manipulating files)

It sounded like this problem may have gone away in an IIS update since Jam wasn't able to reproduce the problem with IIS 4 or 5. If this was just an IIS bug that was fixed in IIS 4, then hopefully it won't be a problem for COD.

Try to reproduce the problem with COD. If it is a problem, it will get interesting since we'll have to use JNI for lower-level win32 file handling. Here's an ongoing email thread on the issue:

```
-----Original Message-----
From: Rajkumar, Isaac
Sent: Friday, January 18, 2002 3:12 PM
To: Afshar, Jamshid; Caldwell, David
Subject: RE: AFM and COD

I was just commenting on the reproducibility - we need to keep track of this for
COD (like David has done). Thanks.

-----Original Message-----
From: Rajkumar, Isaac
Sent: Friday, January 18, 2002 3:09 PM
To: Afshar, Jamshid; Caldwell, David
Subject: RE: AFM and COD

This fix was in 5.0.x and lost in 5.5. - that is over 1-1/2 years of 5.5. and this
is the first time I am hearing about this. So, it looks like this is a pretty rare
occurrence that is not easily reproducible.

-----Original Message-----
From: Afshar, Jamshid
Sent: Friday, January 18, 2002 3:04 PM
To: Caldwell, David; Rajkumar, Isaac
Subject: RE: AFM and COD

Although I wasn't able to reproduce, a customer is seeing what appears to be the
same problem (we have Quickfix 59939 for Banco De Calicia). Also, I just heard from
Mark that QA is able to reproduce it. I'll let you know what happens...
```

**Resolution:**

Instance version, instance state, and the instance cleanup thread obviate this problem.

## *14.9 WebServerLookup Concurrency*

### 14.9.1 Concurrent placement-msgs

If 2 PGs concurrently generate a dynamic or cacheable page, 2 threads can end up trying to add the same WebServerLookup row at the same time. The solution is to check for existence of the row before inserting it, where both the check and the insert are in the same transaction.

### 14.9.2 Concurrent template delete metadata-msg and placement-msg

If a PG generates a cached page at the same time a template delete metadata-msg occurs, 2 threads can end up trying to insert and remove WebServerLookup rows for the same template at the same time. The solution is for PlacementMsgHandler to check for existence of the template metadata in the same transaction that inserts the WS lookup row. If the template metadata doesn't exist, the lookup insert is aborted. Furthermore, MetadataMsgHandler deletes the WS lookup data for the template in the same transaction that deletes the template metadata.

### 14.9.3 Concurrent template add metadata-msg and placement-msg

If a PG generates a dynamic page at the same time a template add metadata-msg occurs, 2 threads can end up trying to insert and remove WebServerLookup rows for the same template at the same time. The solution is for PlacementMsgHandler to check for existence of the template metadata in the same transaction that inserts the WS lookup row. If the template metadata exists, the lookup insert is aborted. Furthermore, MetadataMsgHandler deletes the WS lookup data for the template in the same transaction that adds the template metadata.

## *14.10 Redundant metadata-msgs*

Multiple AMs send metadata-msgs to CM; 1 AM goes down for a while; template changes are deployed; CM clears the cache appropriately; lots of new instances are cached; AM comes back up & starts sending the "old" template update messages to CM.

Here is a solution that prevents CM from repeating the clear cache operations unnecessarily (tMd = template metadata, mdMsg = metadata message):

1. Add objectId and modCount columns to template metadata
2. AM includes objectId and modCount in template add/update/delete metadata-msgs to CM (In exploded deployment these come from the template itself. For templates in a war file, they are copies of the war file's objectId and modCount).
3. CM.MetadataMsgHandler.addTemplate:
    a. if tMd exists, return
    b. add the template md, etc
4. CM.deleteTemplate:
    a. if tMd does not exist, return
    b. if tMd.objectId != mdMsg.objectId, return
    c. delete the template md, etc

5. CM.updateTemplate:
    a. if tMd does not exist, return
    b. if tMd.objectId != mdMsg.objectId, return
    c. if tMd.modCount >= mdMsg.modCount, return
    d. if tMd.templateHash == mdMsg.templateHash, return
    e. update template md, etc
6. CM.updateTemplateMetadata - same as updateTemplate except step 4 compares metadataHash instead of templateHash
7. CM stores mdMsg objectId and modCount in tMd when it adds or updates the template or template md

Why is objectId required? Object id becomes important when a template in exploded deployment, or an entire war file otherwise, is created, updated several times, deleted, then re-added. If an AM goes down before the deletion, when it comes back up and sends CM the delete with a high modCount, if there's no objectId, it appears to be a new message since the modCount saved in the metadata went back to 1 when the template or war was re-added.

Why do updateTemplate and updateTemplateMetadata check the hash if the modCount is ok? Here's the scenario that uses the hash:

app.war created with objectId=m, modCount=1. It contains:

a.jsp with hash=a

b.jsp with hash=b

This results in the following metadata:

a.jsp with objectId=m, modCount=1, hash=a

b.jsp with objectId=m, modCount=1, hash=b

app.war updated with objectId=m, modCount=2. It contains:

a.jsp with hash=c

b.jsp with hash=b

This results in the following metadata:

a.jsp with objectId=m, modCount=2, hash=c

b.jsp with objectId=m, modCount=1, hash=b

Now suppose a new AM is added

app.war is created with objectId=m, modCount=2. It contains:

a.jsp with hash=c

b.jsp with hash=b

AM sends a metadataMsg containing:

a.jsp with objectId=m, modCount=2, hash=c

b.jsp with objectId=m, modCount=2, hash=b

If CM just looked at objectId and modCount, it would unnecessarily clear all of b.jsp's instances.